

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

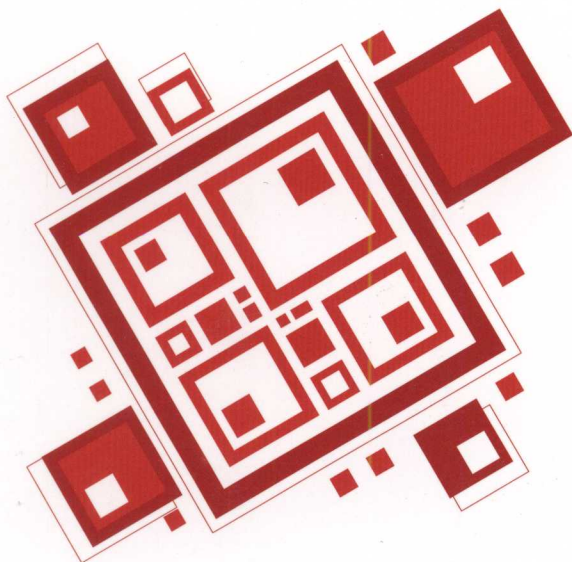
◇ **Laravel**

为Web艺术家创造的PHP“全栈”式框架——简单、优雅、高效！

◆ 深入探究Laravel的艺术性！

◆ 全面解析Laravel的核心点！


Broadview
www.broadview.com.cn



Laravel

框架关键技术解析

陈昊 陈远征 陶业荣 等编著

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Laravel

框架关键技术解析

陈昊 陈远征 陶业荣 魏佩 岁赛 等编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书以Laravel 5.1版本为基础,从框架技术角度介绍Laravel构建的原理,从源代码层次介绍Laravel功能的应用。通过本书的学习,读者能够了解Laravel框架实现的方方面面,完成基于该框架的定制化应用程序开发。

本书第1章到第4章主要介绍了与Laravel框架学习相关的基础部分,读者可以深入了解该框架的设计思想,学习环境搭建,了解PHP语法知识和HTTP协议;第5章到第14章分别从某个方面介绍Laravel框架是如何构建和使用的,包括程序的生命周期、服务容器和数据库等,同时也将其中的一些构建技术剥离开,使读者可以学习该框架的构建技术和思想,如设计模式的内容;第15章是一个简单的实例,将前面的学习内容串联起来并在实践中应用,使读者学会使用该框架定制化地设计应用程序。

本书既适合想了解Laravel框架构建技术的读者,也适合想深入了解Laravel框架的读者。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Laravel框架关键技术解析 / 陈昊等编著. —北京:电子工业出版社, 2016.7

ISBN 978-7-121-29209-5

I. ①L… II. ①陈… III. ①网页制作工具—PHP语言—程序设计 IV. ①TP393.092②TP312

中国版本图书馆CIP数据核字(2016)第146789号

策划编辑:孙学瑛

责任编辑:徐津平

印 刷:北京天宇星印刷厂

装 订:北京天宇星印刷厂

出版发行:电子工业出版社

北京市海淀区万寿路173信箱

邮编:100036

开 本:787×1092 1/16 印张:21 字数:470千字

版 次:2016年7月第1版

印 次:2016年10月第2次印刷

定 价:79.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至zlts@phei.com.cn,盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线:010-51260888-819,faq@phei.com.cn。

Laravel 框架

Laravel 框架是通过 PHP 编程语言编写的，而 PHP 编程语言因为其混乱的设计和优雅的风格有时被认为设计得很糟糕，但是它也有自身的优势，就是专为 Web 开发而生，简单高效是它的法宝，这里的高效不是指它运行的速度快，而是同样的一个任务用它完成的速度快。有统计表明，同样的任务，用 PHP 开发的时间是用 Java 开发的时间的一半左右，因此也诞生了大量用 PHP 编写的 Web 框架（如 Symfony、CodeIgniter、Yii Framework 等）和内容管理系统（如 drupal、Joomla、WordPress 等），通过这些资源可以更加快速地构建 Web 应用。前面提到，PHP 编程语言设计得不是很优美，所以很多用 PHP 编写的框架也比较混乱，而 Laravel 框架开发的宗旨就是为 Web 艺术家创造的 PHP 框架，用糟糕的编程语言设计优雅的框架这一点非常难，但是 Laravel 做到了，对于用户的请求，它就像流水线作业一样，通过一道道工序处理用户的请求，然后返回处理的结果。在这个过程中，用户可以很容易地增加、修改、删除其中的工序，实现定制化。能够做到这些，我想主要是因为开发者在设计期间采用了组件化开发、依赖注入、接口编程等技术，组件化开发使得整个框架像搭积木一样构建起来，因此就可以非常容易地添加、删减功能，体现了编程技术中的易复用、可扩展等特性，依赖注入、接口编程使得模块间的耦合非常低，如果想将某个模块替换自己新设计的模块，只需要满足接口规范就不会对其他模块产生影响，这体现了编程技术中的易维护特性。总之，通过学习 Laravel 框架，不仅可以掌握 Web 开发的方方面面，最重要的是能够学到构建一个优秀框架的思想和方法。

适合的读者

撰写本书的初衷是我在学习 Laravel 框架的过程中看到中文的资料很少，英文资料大多是如何应用该框架，而这些知识不足以支持你创建一个定制化的应用，于是，我花了大量的时间在阅读该框架的源码上，走过了盲人摸象的过程，最后才看到了 Laravel 框架的真面目，这个过程其实是非常低效的。所以，我希望能写一本这样的书，使得以后学习该框架的人能够少走弯路，节省出更多的时间去做更有意义的事。

本书重点介绍了 Laravel 框架构建的关键技术，即组件化开发和使用的相關设计模式，所以本书适合想了解框架构建技术的读者。同时，本书是从源码层次分析该框架实现的几个方面，通过这些源码读者能了解实现的细节，从而很容易实现对该框架的定制和修改，并非只是简单的应用，通过掌握该框架的几个重要方面，读者能够在整体上把握 Laravel 框架实现的过程，所以本书适合想深入了解 Laravel 框架的读者。但是本书并没有提供太多的应用实例，也没有将 Laravel 框架的所有功能进行全部介绍，所以并不适合想快速学习 Laravel 框架应用的读者。

由于本人的能力有限，书中难免有错误、疏漏的地方，敬请大家批评指正。

致谢

首先，最应该感谢该框架的开发者 Taylor Otwell，没有他无私的奉献就没有这么优美的框架供我们使用和学习；其次，感谢我的家人，是你们的支持和理解让这本书在数不清的加班中诞生；感谢参与写作、审校的同事，包括陈远征、陶业荣、魏佩、岁赛、胡凯平、赵艳丽、陈延仓、王鹏、张颂、陈欢、王振全、李鹏飞、赵亚新等；最后，感谢电子工业出版社的孙学瑛编辑，这本书的出版与你们辛勤的努力和负责的态度是分不开的。

第 1 章 组件化开发与 composer 使用	1
1.1 组件化开发	1
1.2 composer 使用	2
1.2.1 composer 安装	2
1.2.2 组件安装	5
1.2.3 自动加载	6
1.2.4 composer 命令行简介	7
1.3 手动构建 Laravel 框架	8
1.3.1 项目初始化	8
1.3.2 添加路由组件	9
1.3.3 添加控制器模块	12
1.3.4 添加模型组件	13
1.3.5 添加视图组件	17
第 2 章 Laravel 框架安装与调试环境建立	21
2.1 Windows 开发环境搭建和调试	21
2.1.1 Laravel 框架安装	21
2.1.2 开发调试环境搭建	22
2.2 Linux 开发环境搭建	31
2.2.1 LAMP 环境搭建	31
2.2.2 Laravel 安装	36

- 第 3 章 Laravel 框架中常用的 PHP 语法 39
 - 3.1 组件化开发语法条件 39
 - 3.1.1 命名空间 39
 - 3.1.2 文件包含 43
 - 3.2 匿名函数 48
 - 3.2.1 匿名函数的使用 49
 - 3.2.2 Laravel 框架中的应用 49
 - 3.3 PHP 中的特殊语法 50
 - 3.3.1 魔术方法 50
 - 3.3.2 魔术常量 53
 - 3.4 反射 55
 - 3.5 后期静态绑定 58
 - 3.6 Laravel 中使用的其他新特性 60
 - 3.6.1 trait 60
 - 3.6.2 简化的三元运算符 63
- 第 4 章 Laravel 框架中使用的 HTTP 协议基础 64
 - 4.1 HTTP 发展与相关网络技术 64
 - 4.1.1 HTTP 发展 64
 - 4.1.2 与 HTTP 协议相关的网络技术 65
 - 4.2 HTTP 协议简介 71
 - 4.2.1 HTTP 协议工作流程 71
 - 4.2.2 请求报文和响应报文结构简介 71
- 第 5 章 Laravel 框架初识 77
 - 5.1 Laravel 框架应用程序目录结构 77
 - 5.1.1 Laravel 框架应用程序根目录介绍 77
 - 5.1.2 app 目录介绍 78
 - 5.1.3 vendor 目录介绍 78
 - 5.2 Laravel 框架应用程序的三个重要环节 79
 - 5.2.1 路由 79

5.2.2 控制器	82
5.2.3 视图	86
第 6 章 Laravel 框架中的设计模式	92
6.1 服务容器	92
6.1.1 依赖与耦合	92
6.1.2 工厂模式	94
6.1.3 IoC 模式	95
6.1.4 源码解析	99
6.2 请求处理管道简介	104
6.2.1 装饰者模式	105
6.2.2 请求处理管道	106
6.2.3 部分源码	110
第 7 章 请求到响应的生命周期	114
7.1 程序启动准备	114
7.1.1 服务容器实例化	115
7.1.2 核心类 (Kernel 类) 实例化	120
7.2 请求实例化	121
7.3 处理请求	124
7.3.1 请求处理准备工作	125
7.3.2 中间件	137
7.3.3 路由处理生成响应	140
7.4 响应的发送与程序终止	146
7.4.1 响应的发送	146
7.4.2 程序终止	148
第 8 章 服务容器与服务提供者	150
8.1 服务容器	150
8.1.1 服务容器的产生	150
8.1.2 服务绑定	151

8.1.3 服务解析	153
8.2 服务提供者	156
8.2.1 创建服务提供者	157
8.2.2 注册服务提供者	158
8.2.3 缓载服务提供者	158
第 9 章 请求与响应的操作	160
9.1 HTTP 请求实例的操作	160
9.1.1 请求实例的获取	160
9.1.2 请求参数的获取	161
9.1.3 请求参数的一次存储	165
9.1.4 获取一次存储数据	166
9.2 HTTP 响应	166
9.2.1 生成响应的主体内容	167
9.2.2 生成自定义响应的实例	167
9.2.3 生成重定向的响应	170
第 10 章 数据库及操作	174
10.1 数据库迁移与填充	174
10.1.1 数据库迁移	174
10.1.2 数据库填充	178
10.2 查询构造器	180
10.2.1 PHP 中数据库的操作	181
10.2.2 数据库连接的封装	185
10.2.3 查询构造器的实现	191
10.2.4 查询构造器的使用	192
10.2.5 查询构造器的数据库操作	196
10.3 Eloquent ORM	198
10.3.1 Eloquent ORM 的底层实现	198
10.3.2 Eloquent ORM 的使用	205

第 11 章	redis 数据库	214
11.1	redis 数据库简介	214
11.1.1	安装	214
11.1.2	redis 数据结构	215
11.2	redis 数据库的应用	220
11.2.1	数据存取	220
11.2.2	redis 数据库编程思想	234
11.2.3	发布、订阅消息	235
第 12 章	会话	239
12.1	Cookie 技术	239
12.2	session 技术	241
12.2.1	session 的工作机制	241
12.2.2	session 的配置	244
12.3	Laravel 框架中的 session 机制	245
12.3.1	session 的启动	246
12.3.2	session 的操作	252
12.3.3	session 的关闭	253
第 13 章	消息队列	256
13.1	同步类型消息队列	257
13.1.1	消息发送	257
13.1.2	消息处理	265
13.2	数据库类型消息队列	267
13.2.1	参数配置	267
13.2.2	数据表的建立	267
13.2.3	消息的生成、发送与处理	269
13.2.4	消息存储	269
13.2.5	消息获取	270

第 14 章 认证与数据验证 273

14.1 认证 273

14.1.1 认证模块的配置 273

14.1.2 数据表的建立 274

14.1.3 添加用户认证路由 275

14.1.4 认证视图的创建 276

14.1.5 用户权限认证 279

14.2 数据验证 285

14.2.1 数据验证的实现 285

14.2.2 数据验证的其他使用方法 289

14.2.3 数据验证后期处理 290

14.2.4 数据验证准则 291

第 15 章 思维笔记实例 293

15.1 数据库设计 293

15.1.1 数据表设计 293

15.1.2 模型类设计 296

15.2 路由设计 301

15.2.1 模块划分 301

15.2.2 程序设计 301

15.3 控制器设计与 Web 页面设计 301

15.3.1 用户认证模块 302

15.3.2 用户管理模块 307

15.3.3 笔记类别管理模块 311

15.3.4 笔记管理模块 317

第1章

组件化开发与 composer 使用

在“敏捷开发”、“不要重复发明轮子”等软件开发思想盛行的当今社会，项目开发中以框架为基础进行二次开发已经成为首选的开发方式，而选取框架的优劣不仅决定了开发的速度，更决定了后期扩展的能力。每一种编程语言都有它的适用范围，其中，PHP 编程语言作为针对 Web 开发量身定制的脚本语言被广泛用于服务器端程序开发，因此也产生了许许多多的 PHP 框架，最著名的有 Laravel、Symfony2、CodeIgniter、Yii2 等，而每年一度的 SitePoint 框架人气调查为这些框架的受欢迎程度提供了数据支持，Laravel 框架已经多年以高出一大截的优势排在第一的位置。Laravel 框架之所以优秀是和它的设计理念分不开的，在 Laravel 官网的首页会看到这样一句介绍的话——为 Web 艺术家创造的 PHP 框架。是的，Laravel 框架的设计理念就是艺术，但设计一个艺术性的框架是离不开那些优秀的设计方法的，这些设计方法虽然算不上创新的方法，但在 Laravel 框架中却得到了恰当的应用，将这些优秀的设计思想和设计方法完美地融合在一起进而产生了 Laravel 的艺术性。如果说哪些设计思想和设计方法是该框架应用比较突出的，笔者觉得至少要包括组件化开发、IoC 容器技术、分布式应用架构设计这三部分内容。对于后两项技术将在后续章节中逐步谈及，本章主要讲解组件化开发思想和 composer 工具的使用。

1.1 组件化开发

组件化开发思想其实在面向过程编程中就已经在使用了，特别对于一些大型的项目，如果不使用组件化的开发思想是很难保证快速开发的。后来面向对象编程思想逐渐流行，由于面向对象本身就隐含着组件化的思想，很多软件项目都是以类为单位进行封装的，所以在一段时间内组件化开发的思想被淡化。但随着软件项目的大型化，开源文化的流行，特别是“不要重复发明轮子”等开源思想的影响，组件化开发再次爆发出强大的力量。在开源文化流行之前，大型项目都是以公司这种集中形式开发的，而相应的代码也很少能够被其他人员重复利用，所以组件化开发并不是必需的。但是，开源文化流行之后，直接利用别人的成果快速构建项目成为一种新的开发模式，而组件化开发方法是这种开发模式的基础。目前，大多数

语言都支持组件化开发,特别是开源的编程语言在这方面支持更好,如 Ruby、Java、Python 等。

组件化开发的目的是能够快速使用已有的程序模块构建项目,甚至可以快速更换项目中的相应模块而不需要修改系统中其他部分的代码,这就需要所有的代码按照一定的规范和接口来实现。首先介绍 PHP-FIG (PHP Framework Interop Group, PHP 框架互动群),它的作用就是制定一系列 PHP 开发的规范即 PSR 编码规范标准,程序员在开发 PHP 程序时共同遵守这个标准,依据此标准开发的组件可以很容易地组合到一起,也可以很方便地被别人使用,也就实现了组件化的管理和开发。读者可以在网站 www.php-fig.org 中找到相关介绍。目前 PSR 标准主要包含 PSR-0~PSR-4 文档,其中最主要的是 PSR-0 和 PSR-4,这两个文档主要制定了 Autoloader 标准即代码自动加载标准,后面章节还会详细介绍这部分内容。

在 Laravel 框架中很多组件也不是重新开发的,而是使用已经存在的其他组件或框架的部分,如 Laravel 底层就使用了很多 Symfony 框架中的组件。在使用 Laravel 框架的过程中,如果对一些组件不满意,可以使用其他开源的组件进行替代或修改,而且几乎不用修改框架中的其他部分代码就能保证各模块间协调工作,这就是规范和接口的威力。

1.2 composer 使用

上一节介绍了组件化开发的优秀特性,那么如何实现组件化开发呢?这里就需要借助一个工具即组件管理工具。在组件化开发中,组件管理工具是必不可少的,因为全世界有大量的组件,如果没有统一的管理工具,只靠人的经验去查找使用,就无法做到统一的管理和控制。不同编程语言的组件化管理工具是不同的,如 PHP 的组件化管理工具是 composer、Ruby 的组件化管理工具是 gem、Java 的组件化管理工具是 maven、Python 的组件化管理工具是 pip 等。下面介绍如何使用 composer 工具实现组件化开发。

1.2.1 composer 安装

不同的操作系统安装 composer 的方式有些不同,这里主要介绍 Windows 系统和 Linux 系统的 composer 安装,首先介绍 Windows 系统中 composer 的安装。

1. Windows 系统环境下安装 composer 工具

composer 工具实际是用 PHP 语言开发的工具,安装 composer 工具需要具有 PHP 环境,在 Windows 系统中可以使用 phpStudy 等集成开发环境,在集成开发环境中安装相应的 PHP 环境,对于 phpStudy 环境的安装可以参考第二章内容。登录 composer 官方网址 <https://getcomposer.org>,在 Download 选项页面中找到 Composer-Setup.exe 并下载,然后直接双击安装。在安装过程中,有两个需要说明的安装选项,分别如图 1.1 和图 1.2 所示。

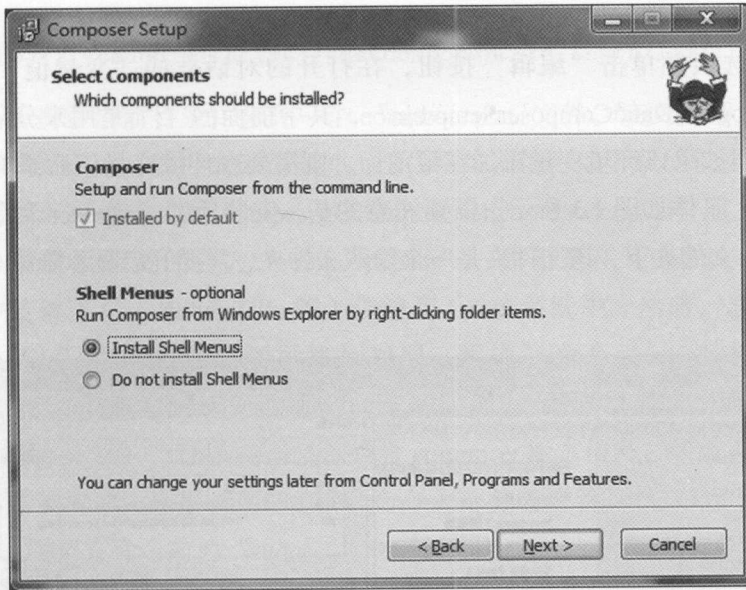


图 1.1 Shell Menus 选项设置

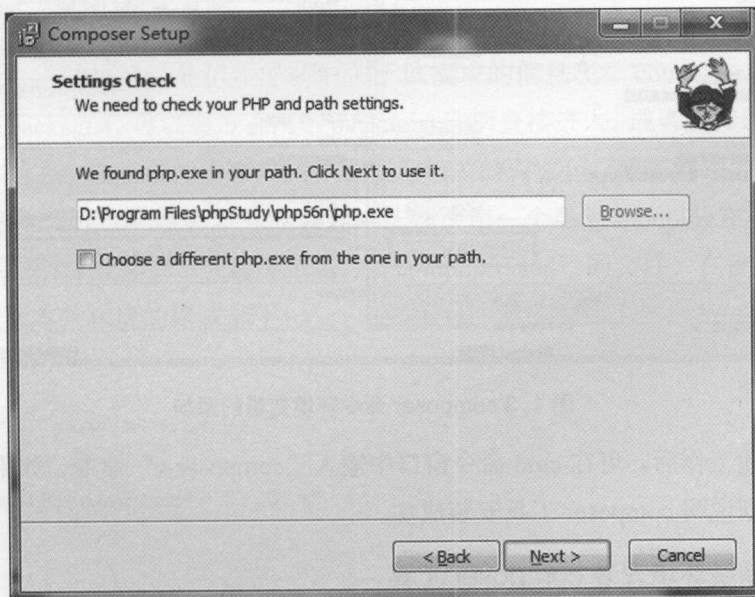


图 1.2 PHP 环境路径选项

在图 1.1 中，默认 Shell Menus 选项是不安装，这里选择安装，选择安装该选项后，可以在任意文件夹下通过单击鼠标右键的方式直接选择使用 composer 工具，而不需要在命令窗口中一步步进入到相应的目录下再使用。在图 1.2 中，PHP 环境路径选项中默认会识别一个 php.exe，如果没有检测到 PHP 环境则需要手动查找。安装完成后还需要在环境变量中添加 composer 的命令目录，这里以 Windows 7 系统为例，用鼠标右键单击“计算机”后选择“属性”，单击左侧列表中的“高级系统设置”，打开“系统属性”对话框，在“高级”选

项卡中单击“环境变量”按钮，打开“环境变量”对话框，在“系统变量”的变量列表中选择“Path”选项并单击“编辑”按钮，在打开的对话框的“变量值”文本框的最后添加“; C:\ProgramData\ComposerSetup\bin”，其中前面的“;”是用来分隔其他环境变量的，不能省略，如果该环境变量添加在最前面，则需要在环境变量值后添加“;”用以与其他的分隔开来，具体如图 1.3 所示。需要注意的是，安装后的 composer 命令路径是在 C 盘的 ProgramData 文件夹下，该文件夹是一个隐藏文件夹，需要开启显示隐藏文件及文件夹功能才能看到。

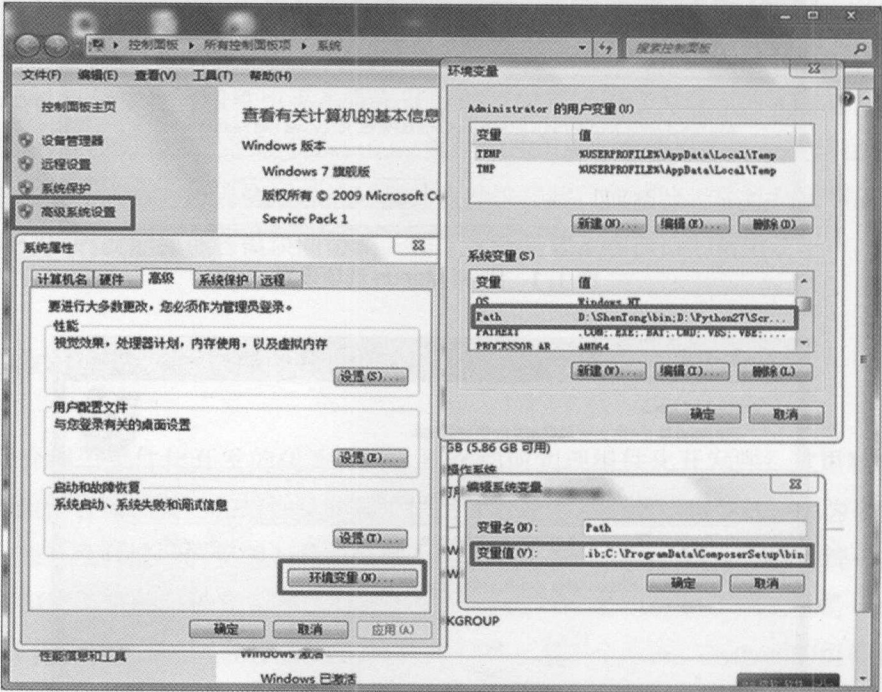


图 1.3 composer 命令环境变量的添加

在完成上述工作后，可在 cmd 命令窗口中输入“composer-v”命令，如果显示 composer 的版本信息，则说明 composer 工具安装成功。

2. Linux 系统环境安装 composer 工具

在 Linux 系统 (这里以 Ubuntu 系统为例) 中安装 composer 工具也需要 PHP 环境，首先通过“apt-get update”命令更新系统的软件列表，然后通过“apt-get install php5”命令来安装 PHP 环境，安装完成后通过“php -v”命令检测 PHP 的版本。安装 composer 需要 PHP 的版本在 5.3 以上，由于 composer 工具管理的组件资源很多需要 PHP 版本在 5.4 以上，因此建议系统中的 PHP 环境版本在 5.4 以上。安装完 PHP 环境后，需要通过“apt-get install curl”命令来安装 curl 工具，该工具相当于一个命令行版的浏览器，可以支持大部分的浏览器功能，有些版本的 Linux 系统默认没有安装该工具。在完成 PHP 和 curl 工具的安装后，可以通过如下两个命令安装 composer。

命令一: `curl -sS https://getcomposer.org/installer | php`

命令二: `php -r "readfile('https://getcomposer.org/installer');" | php`

安装成功后查看当前目录, 会有一个 `composer.phar` 文件, 该文件即为 `composer` 工具执行文件。由于 `composer` 软件的源在国外, 国内网络环境可能下载比较慢或者根本就无法下载, 因此读者可以通过网址 “<https://getcomposer.org/composer.phar>” 直接将 `composer` 工具进行下载, 下载后的文件名为 “`composer.phar`”。如果是在 Windows 系统中下载的, 可以通过文件传输工具 (如 FileZilla 等) 传输到 Linux 系统中。然后, 通过命令 “`chmod +x ./composer.phar`” 为该文件添加可执行权限, 再通过命令 “`mv composer.phar /bin/composer`” 将文件 `composer.phar` 移动到 “`/bin`” 目录下, 并修改文件名为 `composer`。同样, 完成后可以直接输入命令 “`composer -v`” 查看 `composer` 版本信息, 如果正确显示, 则说明 `composer` 工具安装成功。这里所用的 Linux 命令大部分需要管理员权限, 如果不是以管理员身份登录的, 需要在命令前加 “`sudo`”。

1.2.2 组件安装

在完成 `composer` 工具的安装后, 就可以通过组件化的方式创建项目了。如果需要在项目添加一个日志的模块, 如何才能获取关于该模块的信息呢? `composer` 官方网址提供了组件资源库即 `packagist`, 可以通过官网的 `packagist` 选项直接进入, 或者直接通过网址 “<https://packagist.org>” 进入, 在资源库中可以浏览或搜索相关的资源包, 如需要一个日志的资源包, 可以通过关键字 “`log`” 来搜索。假设项目中选择使用一个名为 `monolog` 的组件来完成日志功能, 则需要在项目根目录下创建一个名为 “`composer.json`” 的文件, 在该文件中记录所需要的组件名及版本, 相应的格式如下:

```
{
  "name": "glow/model-test",
  "require": {
    "monolog/monolog": "1.0.*"
  }
}
```

这里包含两个标签, 其中 “`name`” 标签表示本项目的名称, “`glow`” 是公司名, 而 “`model-test`” 是项目名称。这个标签不是必需的, 但是如果将自己的项目作为一个资源包发布就需要这个名字, 使得其他人可以通过该名称下载这个组件。 “`require`” 标签表示需要的资源包, 其中 “`monolog/monolog`” 为资源包的名称, “`1.0.*`” 为版本号, 这里的版本号可以通过几种方式约束, 分别是确切版本号 (如 `1.1.1`)、范围版本号 (如 `>=1.1`、`<2.3` 等)、通配符版本号 (如 `1.0.*`, 用于匹配 `>=1.0` 并且 `<1.1` 的版本) 和赋值运算版本号 (如 `~1.0`, 用于匹配 `>=1.0` 并且 `<2.0` 的版本)。

创建完 `composer.json` 文件后, 通过 `cmd` 命令窗口进入到项目的根目录, 如果在安装

composer 时选择安装了“Shell Menus”选项，则可以在项目的根目录中单击鼠标右键并选择“use composer here”选项，在命令窗口中输入命令“composer install”，接着 composer 会检查 composer.json 文件中的组件名称及版本，将它下载到当前目录的 vendor 文件夹下，如果当前目录下没有 vendor 文件夹，将会自动创建一个。对于上面实例，创建的目录结构为“vendor\monolog\monolog”。在完成组件下载后，会在当前目录下创建一个名为“composer.lock”的锁文件，该文件将记录当前项目依赖组件的确切版本号，当执行“composer install”命令时会首先查看该文件中的版本，如果存在则下载该文件中指定的版本。这点对于分布式开发非常有用，不同开发人员只需要上传 compsoer.lock 文件到版本库，其他人通过该文件就可以下载相同版本的组件，实现程序的版本统一。如果某组件有了更新的版本，需要更新组件，可以通过“composer update”命令实现。如果这部分内容还不能很好的理解，不要紧，后面将会有有一个实例让读者亲手实践，那时就会掌握并理解这些命令的用法。

1.2.3 自动加载

通过 composer 的 install 命令除了可以下载组件以外，还会在 vendor 目录下提供一个自动加载文件，只需要在项目中通过“require 'vendor/autoload.php';”语句引入这个文件，在使用下载的组件时就可以实现自动加载了。如上一小节的实例中，下载了 monolog 组件，就可以通过“\$myLog = new \monolog\Logger('wangshuo')”语句直接使用组件中的类库，而 autoload 文件会自动加载相应的类文件。

实现文件自动加载需要有相应的规范进行约束，其中包括 PSR-0、PSR-4、classmap 和 files 四种规范形式，其中 PSR-4 是目前推荐使用的规范。这四种规范形式本质上是定义了一个命名空间到实际文件的映射关系，通过这个映射关系，可以利用命名空间类精确定位到相应文件的具体路径，进而实现“autoload”自动加载功能。首先介绍 PSR-0 和 PSR-4 规范，这两种规范相似。在 composer.json 文件中可以直接添加 autoload 字段实现命名空间到目录的映射，如 Laravel 框架中 APP 命名空间下类的自动加载设置如下：

```
{
    "autoload": {
        "psr-4": { "App\\" : "app/" },
        "psr-0": { "Bpp\\" : "bpp/" }
    }
}
```

这里根据 PSR-0 和 PSR-4 规范定义了两个映射关系，即命名空间“App\\”对应目录“app/”和命名空间“Bpp\\”对应目录“bpp/”。在 PSR-4 规范下，假设创建一个 app/User.php 文件，则该文件需要包含 App\User 类，也就是说当使用“\$user = new \App\User()”语句实例化 App\User 类时，autoload 会根据定义的 PSR-4 规范到目录 app/ 下查找 User.php 文件；在 PSR-0 规范下，则需要创建一个 bpp/Bpp/User.php 文件，而该文件中包含 Bpp\User 类。这

里的区别就在于 PSR-4 规范的目录不需要添加命名空间 “App”，而 PSR-0 规范的目录需要添加命名空间 “Bpp”。

对于 classmap 会扫描指定目录中所有的 .php 和 .inc 文件，并加载到 vendor/composer/autoload_classmap.php 文件中，在该文件中会实现一个具体类与文件映射的关联数组，也可以直接精确指定一个文件。通过 classmap 可以生成不遵循 PSR-0 和 PSR-4 规范的自动加载类库。对于下面的实例，就会搜索 database 目录下的所有 .php 文件和 .inc 文件，并记录类名与文件的对应关系。

```
{
  "autoload": {
    "classmap": [ "database" ]
  }
}
```

对于在每次程序执行时都需要载入的文件，可以通过 files 规范实现自动加载，通常经常使用的函数库文件就使用这种载入方式，例如下面的实例每次都会加载。

```
{
  "autoload": {
    "files": [
      "src/Illuminate/Foundation/helpers.php",
      "src/Illuminate/Support/helpers.php"
    ]
  }
}
```

1.2.4 composer 命令行简介

在前面的介绍中已经接触到一些 composer 命令，表 1.1 对常用的命令进行了简单的归纳，如果需要更加详细地了解可以参看 composer 官网。

表 1.1 composer 工具常用命令功能

命 令	功 能
composer list	获取帮助信息
composer init	以交互方式填写 composer.json 文件信息
composer install	从当前目录读取 composer.json 文件，处理依赖关系，并安装到 vendor 目录下
composer update	获取依赖的最新版本，升级 composer.lock 文件
composer require	添加新的依赖包到 composer.json 文件中并执行更新
composer search	在当前项目中搜索依赖包
composer show	列举所有可用的资源包

续表

命 令	功 能
composer validate	检测 composer.json 文件是否有效
composer self-update	将 composer 工具更新到最新版本
composer create-project	基于 composer 创建一个新的项目
composer dump-autoload	在添加新的类和目录映射时更新 autoloader

1.3 手动构建 Laravel 框架

在学习了组件化开发思想和 composer 资源包管理工具的使用后，读者已经具备了组件化搭建大型项目的基本能力。本节将带领读者一步步搭建一个类似于 Laravel 的服务器端程序框架，该框架包含服务器端程序中的 MVC（模型、视图和控制）三个组成部分，可以实现服务器端程序中的路由、控制器、数据库操作及视图模板等主要功能。该框架使用的组件都是 Laravel 中使用的组件，读者目前不用深究实现过程中的一些细节，当学完后面章节中关于 Laravel 框架各模块的关键技术时，对于本节中实现的细节也会随即掌握。构建过程分为项目初始化、路由组件添加、控制器模块添加、模型组件添加和视图组件添加五个步骤，当通过这五个步骤完成一个具有现代意义的服务器端程序框架时，你会发现原来程序设计可以如此神奇和艺术。

1.3.1 项目初始化

对于这个类 Laravel 项目，暂且起名为 lara。首先在服务器的 web 目录下创建一个 lara 文件夹，本实例中创建的路径为“D:\WWW\lara\”，上述路径作为网站的根目录，在根目录下新建一个“composer.json”文件，并输入如下内容：

文件 D:\WWW\lara\composer.json

```
{
    "require": {
    }
}
```

通过 composer 工具使用的介绍，知道该文件为资源包管理文件，此时创建的文件没有添加任何依赖包，即“require”项中的值为空。文件创建完成后，通过 cmd 命令行在 lara 目录下运行“composer update”命令，执行成功后 lara 文件夹下会自动生成自动加载文件，其目录结构和文件如图 1.4 所示。这些与自动加载相关的目录和文件都是 composer 工具自动生成的，对于自动加载的具体实现过程将在后续章节中详细介绍。

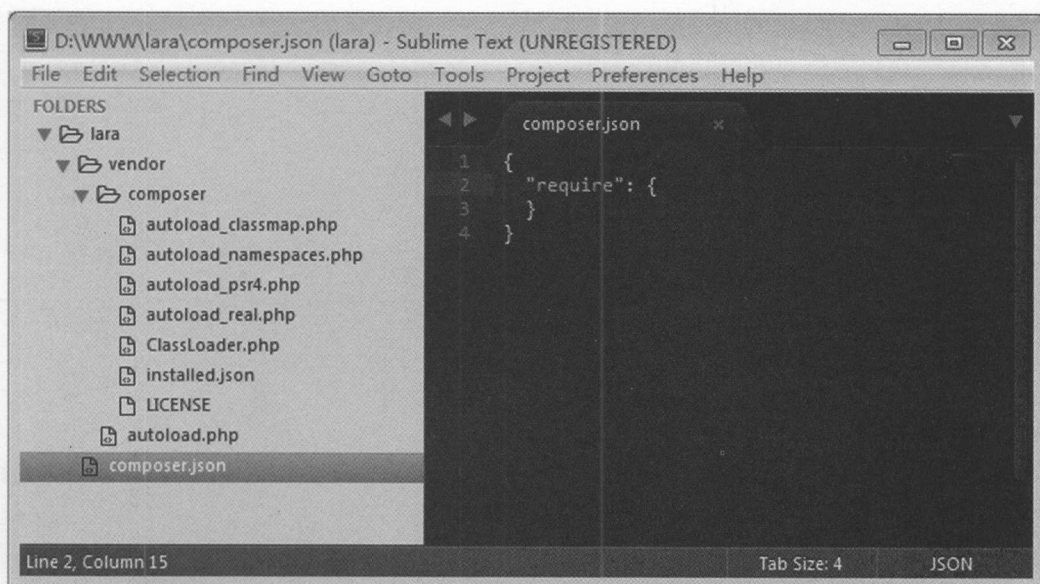


图 1.4 自动加载的目录结构及文件

1.3.2 添加路由组件

在完成了项目初始化后，将进行第一个组件的添加，即路由组件的添加。首先登录 composer 官网，网址是“<https://getcomposer.org>”，选择 Packagist 选项，Packagist 是 composer 工具的主要资源包管理库，在搜索框中输入“route”可以查找到很多关于路由的组件，其中可以看到组件名“illuminate/routing”，单击可以查看关于该组件的详细信息。这里准备使用的路由组件就是该组件，也是 Laravel 框架中使用的组件。需要注意的是，该组件中也会有一个“composer.json”的文件，该文件会记录该组件所依赖的其他组件，包括“symfony/routing”和“illuminate/container”等，但是该组件还有一个“illuminate/events”组件没有包含，需要在添加路由组件时一起添加。于是，需要修改 lara 目录下的“composer.json”文件为如下内容：

文件 D:\WWW\lara\composer.json

```

{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*"
  }
}
```

修改后依然执行“composer update”命令，完成两个组件及其依赖组件的下载，下载完成后的目录结构及文件如图 1.5 所示。

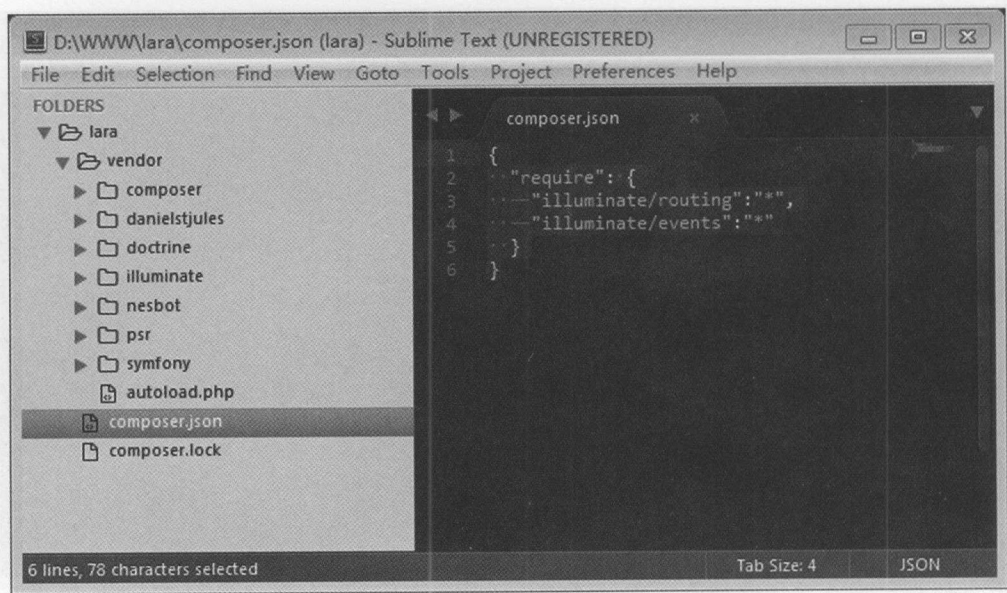


图 1.5 完成路由组件添加后的目录结构及文件

在完成路由组件的添加后该如何使用呢？首先需要添加两个文件，一个是路由文件，另一个是服务器端程序入口文件。需要注意的是，本实例中将按照 Laravel 框架的目录结构来添加相关文件，主要是帮助读者熟悉 Laravel 框架的目录结构，为以后学习 Laravel 框架打下基础。于是，对于路由配置文件，需要在 lara 目录下创建一个 app 目录，该目录主要存储项目开发的文件，在该目录下再创建一个 Http 目录，该目录用于存储处理 HTTP 请求的文件，在 Http 目录下再创建一个 routes.php 文件，该文件就是所要创建的路由文件，在 Laravel 框架中路由文件也在这个目录结构下。添加的路由文件主要代码如下：

文件 D:\WWW\lara\app\Http\routes.php

```
<?php
$app['router']->get('/', function() {
    return '<h1>路由成功!!! </h1>';
});
```

接下来将添加服务器端程序请求入口文件，首先在 lara 目录下创建一个 public 目录，该目录用于存放项目的公共文件，即通过 HTTP 请求可以访问到的文件，包括入口文件、JS 文件和 CSS 文件等，在该目录下创建一个 index.php 文件，该文件即为服务器端程序请求入口文件，在 Laravel 框架中入口文件也在相同的目录结构下。添加的入口文件内容如下：

文件 D:\WWW\lara\public\index.php

```
<?php
// 调用自动加载文件，添加自动加载文件函数
require __DIR__.'../../vendor/autoload.php';
// 实例化服务容器，注册事件、路由服务提供者
```

```

$app = new Illuminate\Container\Container;
with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();
// 加载路由
require __DIR__.'/../app/Http/routes.php';
// 实例化请求并分发处理请求
$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
// 返回请求响应
$response->send();

```

在完成上述两个文件的添加后，通过访问该网站根目录可以得到相应的路由响应，响应输出如图 1.6 所示。在本实例中使用的是 PHP 集成开发环境 phpStudy，对于环境安装的介绍请参照后面章节。

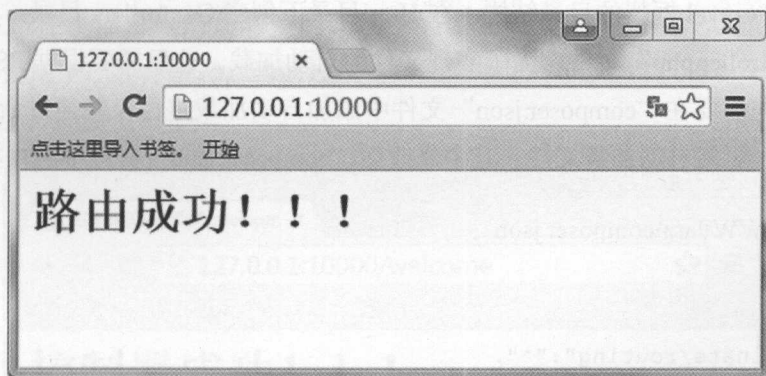


图 1.6 路由组件响应

下面简要介绍两个文件实现的功能，只需要了解就好，在后续章节这些内容都会详细介绍。请求访问的入口文件“index.php”主要完成了几部分工作，分别是自动加载函数的添加、服务容器实例化与服务注册、路由加载、请求实例化与路由分发、响应生成与发送。其中，自动加载函数用于自动包含引用文件，所以要首先添加。前文也提到过，该文件是 composer 工具自动生成的，直接通过 require 关键字添加即可。在 Laravel 框架中一些功能的生成都需要服务容器来实现，即 Illuminate\Container\Container 类的实例，服务容器用于服务注册和解析，也就是说向服务容器注册能够实现某些功能的实例或回调函数，当需要使用该功能时从服务容器中获取相应的实例来完成。本实例中，在完成了服务容器实例化后进行了事件和路由的服务注册，然后通过服务容器获取路由处理的相关实例并完成路由的加载和请求处理，而获取路由实例是通过代码“\$app['router]”实现的，其中“\$app”为服务容器实例，而“[router]”则表示服务容器中注册的路由服务名称。在路由文件“routes.php”中，通过路由实例的 get() 方法添加了一条路由，其中路由名称为“/”，即表示网站根目录，路由的处理函数为一个匿名函数，用于返回响应。当通过浏览器访问地址“http://127.0.0.1:10000”时，即访问网站的根目录，服务器会调用入口文件进行处理，入口文件会通过 Illuminate\

HttpRequest 类的静态方法 createFromGlobals() 实现请求的实例化，然后通过路由进行分发处理，路由会根据请求的地址查找路由表，查找到的通过路由表中对应的相应函数进行请求处理并返回响应，否则将返回页面显示未找到的 404 错误。以上就是路由的基本工作过程。

1.3.3 添加控制器模块

前面完成了路由组件的添加，可以通过匿名函数实现请求的处理响应。但是，在实际项目中请求的处理往往比较复杂，如果将处理函数写在路由文件中会显得混乱，也不容易管理，因此会将路由的处理部分单独用控制器类来实现。其实，在添加路由组件时已经添加了基本控制器类，即 Illuminate\Routing\Controller 类，在添加控制器模块时可以使用该类作为基类以扩展控制器的功能，本实例中为了使不同模块间更加清晰，将不使用该类作为基类，而是直接创建控制器类。

这里依照 Laravel 框架的目录结构，在 Http 目录下创建 Controllers 目录，并在其中添加 “WelcomeController.php” 文件。为了实现文件的自动加载，这里需要根据 PSR-4 规范进行相关配置，首先需要在 “composer.json” 文件中添加自动加载路径，并通过命令 “composer dump-autoload” 更新自动加载文件。其中修改后 “composer.json” 文件中的内容如下：

文件 D:\WWW\lara\composer.json

```
{
    "require": {
        "illuminate/routing": "*",
        "illuminate/events": "*",
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/"
        }
    }
}
```

在完成自动加载文件的更新后，其实控制器就可以使用了。为了验证控制器的功能，添加路由配置和控制器处理函数分别如下：

文件 D:\WWW\lara\app\Http\routes.php

```
<?php
$app['router']->get('/', function() {
    return '<h1>路由成功!!! </h1>';
});
$app['router']->get('welcome',
    'App\Http\Controllers>WelcomeController@index');
```

文件 D:\WWW\lara\app\Http\Controllers>WelcomeController.php

```
<?php
namespace App\Http\Controllers;
class WelcomeController
{
    public function index()
    {
        return '<h1> 控制器成功!!! </h1>';
    }
}
```

在路由配置文件“routes.php”中添加了一条路由配置，即当请求的 URL 为 welcome 时，由“App\Http\Controllers>WelcomeController@index”来处理，其中 App\Http\Controllers>WelcomeController 为控制器类，而 index 为控制器类的处理函数名称。对于控制器类文件需要按照文件路径添加正确的命名空间，其中类名要与文件名相同，这些内容在前文的 composer 工具使用中已经介绍过。当通过浏览器访问地址“http://127.0.0.1:10000/welcome”时，将获得响应，如图 1.7 所示。

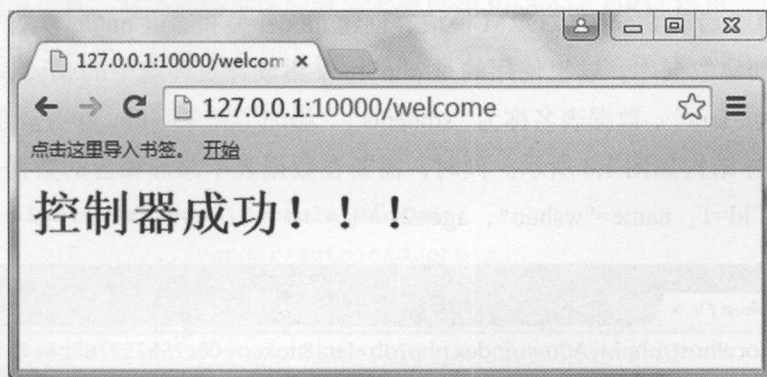


图 1.7 控制器响应

1.3.4 添加模型组件

控制器中没有添加相应的组件，因为比较简单，而且在路由组件中也有相应的功能，所以只是实现了模块化添加。下面将添加模型组件，模型组件相当于 MVC 中的 M，主要实现数据处理功能，这部分功能将使用 Laravel 框架中的“illuminate/database”组件来完成。依然可以通过 composer 官网的 Packagist 资源包管理器搜索“database”来查看该组件的相关内容，在资源包管理器中，可以看到很多具有类似功能的组件，如果读者喜欢，完全可以使用其他的组件来代替。在添加模型组件前，首先依然需要修改“composer.json”文件，通过 composer 工具来帮助下载并解决依赖关系，然后执行“composer update”命令，完成模型组件及其依赖的下载，其中修改后的“composer.json”文件内容如下：

文件 D:\WWW\lara\composer.json

```
{
    "require": {
        "illuminate/routing": "*",
        "illuminate/events": "*",
        "illuminate/database": "*",
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/"
        }
    }
}
```

“illuminate/database” 组件主要用于操作数据库，它提供了两种操作数据库的方式，一种是查询构造器方式，另一种是 Eloquent ORM 方式，这里使用 Eloquent ORM 方式，通过该方式操作数据库非常简单，对于这两种方式的实现细节在后面的数据库章节中都有相关介绍。通过 Eloquent ORM 方式操作数据库需要完成以下五步工作，分别是创建数据库、添加数据库配置信息、启动 Eloquent ORM 模块、创建 model 类和通过 model 类操作数据库。

下面首先创建数据库，这里使用的是 phpStudy 集成开发环境中的 MySQL 数据库，其中数据库名称为“lara”，数据表名称为“students”，添加了三个字段，分别是“id”、“name”和“age”，具体结构如图 1.8 所示。同时，需要在数据表中添加相应的数据，这里只添加一条数据，即“id=1, name="wshuo", age=21”。

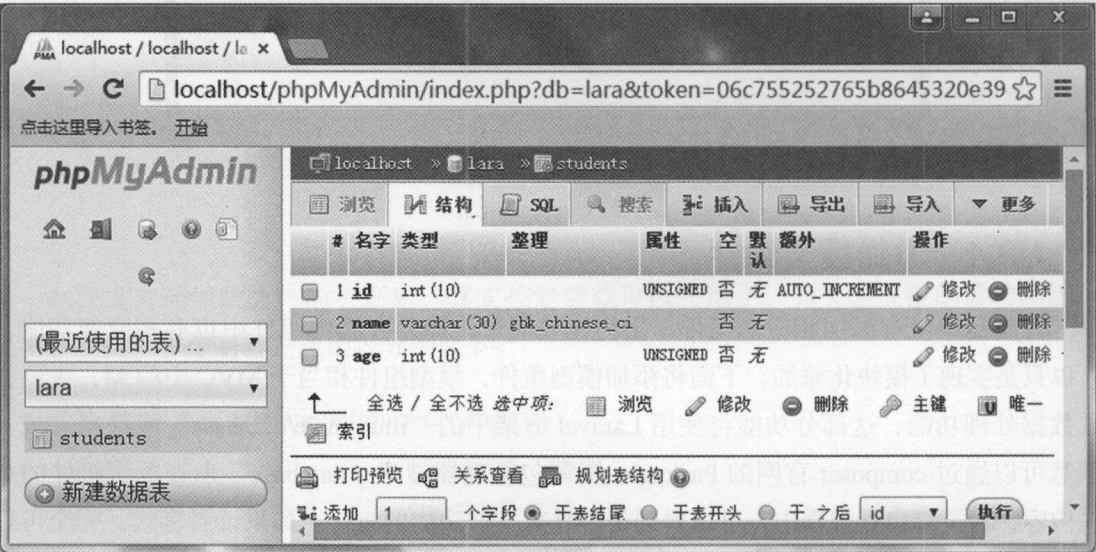


图 1.8 students 数据表结构

在完成数据库的创建及数据的添加后，下面需要添加数据库的配置信息。依然按照

Laravel 框架的目录结构，在 lara 目录下创建 config 文件夹，该文件夹用于存放整个应用程序的配置文件，在该文件夹下创建“database.php”文件，用于存储对数据库的配置信息。其中，配置文件都是以数组的形式提供配置信息的。数据库的配置内容如下：

文件 D:\WWW\lara\config\database.php

```
<?php
return [
    'driver'      => 'mysql',
    'host'        => 'localhost',
    'database'     => 'lara',
    'username'     => 'root',
    'password'     => 'root',
    'charset'      => 'utf8',
    'collation'    => 'utf8_general_ci',
    'prefix'       => ''
];
```

通过数据库配置数组中的内容可以看出，主要配置数据库的主机地址、数据库名称、用户名和密码等信息。在完成数据库配置后需要启动 Eloquent ORM 模块，这部分工作是在访问入口文件中实现的，具体内容如下：

文件 D:\WWW\lara\public\index.php

```
<?php
use Illuminate\Database\Capsule\Manager;

require __DIR__.'../../vendor/autoload.php';
$app = new Illuminate\Container\Container;
with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();
// 启动 Eloquent ORM 模块并进行相关配置
$manager = new Manager();
$manager->addConnection(require '../config/database.php');
$manager->bootEloquent();
require __DIR__.'../../app/Http/routes.php';
$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
$response->send();
```

启动 Eloquent ORM 模块阶段需要用到数据库的管理类，即 Illuminate\Database\Capsule\Manager 类，于是添加了对应的命名空间并进行了初始化，然后通过 addConnection() 函数完成数据库的相关配置并通过 bootEloquent() 函数完成数据库 Eloquent ORM 模块的启动。在启动完成后，就可以操作数据库了。通过 Eloquent ORM 方式操作数据库需要两个步骤来实现，一是创建模型类，二是通过模型类的方法操作数据库。下面就来实现模型类的创建。

模型类的目录结构与 Laravel 框架略微有些差别，Laravel 框架中会将模型类直接放在 app 目录下，这里会在 app 目录下创建一个 Models 文件夹，用于统一管理模型，使程序看起来更加模块化，然后在 Models 文件夹下创建 “Student.php” 文件，在该文件中创建 App\Models\Student 模型类。具体内容如下：

文件 D:\WWW\lara\app\Models\Student.php

```
<?php namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Student extends Model
{
    public $timestamps = false;
}
```

现在模型类创建完成了，其中只有一句代码，接下来学习通过模型类操作数据库。路由依然使用 1.3.3 节添加控制器模块中的路由，只是将控制器中的处理函数 index() 进行了修改，修改后处理函数内容如下：

文件 D:\WWW\lara\app\Http\Controllers>WelcomeController.php

```
<?php namespace App\Http\Controllers;
use App\Models\Student;
class WelcomeController
{
    public function index()
    {
        $student = Student::first();
        $data = $student->getAttributes();
        return "学生 id=" . $data['id'] . "； 学生 name=" .
            $data['name'] . "； 学生 age=" . $data['age'];
    }
}
```

通过 Eloquent ORM 方式操作数据库首先需要引入模型类，这里通过 use 关键字引入 App\Models\Student 模型类，在处理函数中通过 “Student::first();” 获取数据库 “lara” 中数据表 “students” 的第一行数据，该数据会被封装到模型类实例 “\$student” 中，然后通过该实例的 getAttributes() 函数获取相应的数据，最后输出结果。当通过浏览器访问地址 “http://127.0.0.1:10000/welcome” 时，将输出获取的学生数据，如图 1.9 所示。

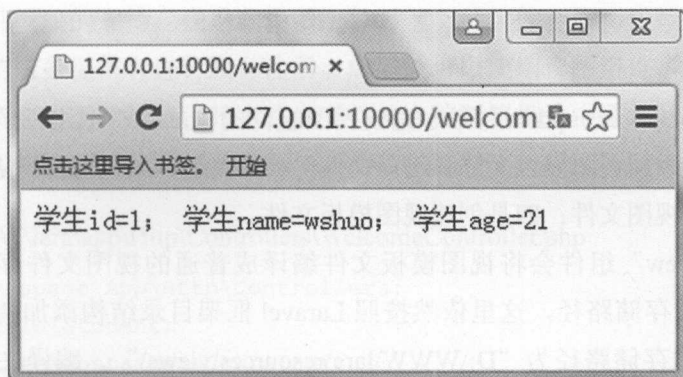


图 1.9 获取数据库数据

至此，模型组件的添加已经完成了。数据库的模型类只有一行代码，而对模型类的操作和显示也只有三行代码，被它的简洁优美震撼了吧。这里只简单介绍一下实现原理，详细介绍可以参看数据库及操作章节。首先，创建的每个模型类都需要继承 Illuminate\Database\Eloquent\Model 类，而每个创建的模型类就对应一个数据表，在默认情况下类名小写后的复数即为对应的数据表名，当通过模型类调用相应方法时，就会操作相应数据表中的数据，本实例中是通过 first() 方法获取数据表的第一行数据。

1.3.5 添加视图组件

在完成模型组件的添加后，接下来将完成最后一个组件的添加，即视图组件的添加。依然使用 Laravel 框架中的视图组件 “illuminate/view”，该组件可以将视图以模板的方式创建，在其他视图中可以调用、继承已经创建的模板，并通过模板语法使得视图设计更加简单、规范、高效。

对于视图组件 “illuminate/view”，可以通过 composer 的 Packagist 资源包管理器来查找，在搜索框中输入 “view” 可以查找到很多视图类的组件。为完成该组件的添加，首先依然需要修改 “composer.json” 文件，修改后的内容如下：

文件 D:\WWW\lara\composer.json

```
{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*",
    "illuminate/database": "*",
    "illuminate/view": "*"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app/"
    }
  }
}
```

```

    }
}

```

接着通过“composer update”命令完成视图组件的添加。视图组件的使用需要完成以下四个工作：一是添加视图模板文件和编译文件的存储路径；二是对视图进行相关配置和服务注册；三是使用视图文件；四是创建视图模板文件。

“illuminate/view”组件会将视图模板文件编译成普通的视图文件存储起来，所以首先需要添加相应的存储路径。这里依然按照 Laravel 框架目录结构添加两个文件路径，其中视图模板文件的存储路径为“D:\WWW\lara\resources\views\”，编译文件的存储路径为“D:\WWW\lara\storage\framework\views\”。接下来完成视图组件的相关配置和服务注册，这部分内容是在程序入口文件中实现的，其内容如下：

文件 D:\WWW\lara\public\index.php

```

<?php
use Illuminate\Database\Capsule\Manager;
use Illuminate\Support\Fluent;
require __DIR__.'../../vendor/autoload.php';
$app = new Illuminate\Container\Container;
Illuminate\Container\Container::setInstance($app);
with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();
// Eloquent ORM
$manager = new Manager();
$manager->addConnection(require '../config/database.php');
$manager->bootEloquent();
$app->instance('config', new Fluent);
$app['config']['view.compiled'] = "D:\\WWW\\lara\\storage\\framework\\
views\\";
$app['config']['view.paths'] = ["D:\\WWW\\lara\\resources\\views\\"];
with(new Illuminate\View\ViewServiceProvider($app))->register();
with(new Illuminate\Filesystem\FilesystemServiceProvider($app))
->register();
require __DIR__.'../../app/Http/routes.php';
$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
$response->send();

```

通过服务容器中的 setInstance() 静态方法将服务容器实例添加为静态属性，这样就可以在任何位置获取服务容器的实例。视图模块的配置稍微复杂一些，首先通过服务容器实例的 instance() 方法将服务名称为“config”和 Illuminate\Support\Fluent 类的实例进行绑定，该类的实例主要用于存储视图模块的配置信息。这里用到的配置信息分别是前文中创建的视图模板文件和编译文件存储路径，分别添加到配置实例中。接下来进行服务注册，因为视图模

块的使用需要文件模块的支持，在下载视图组件时，文件组件会作为依赖下载，所以可以直接使用文件组件的服务提供者进行服务注册。然后就可以使用视图组件了，这里在处理函数中使用视图组件实现视图的加载，其中路由文件与在控制器模块中添加的相同，处理函数依然是 `index()` 函数。通过视图模块返回视图响应的处理函数内容如下：

文件 D:\WWW\lara\app\Http\Controllers\WelcomeController.php

```
<?php namespace App\Http\Controllers;
use App\Models\Student;
use Illuminate\Container\Container;
class WelcomeController
{
    public function index()
    {
        $student = Student::first();
        $data = $student->getAttributes();
        $app = Container::getInstance();
        $factory = $app->make('view');
        return $factory->make('welcome')->with('data',$data);
    }
}
```

首先，通过服务容器的 `getInstance()` 静态方法获取服务容器实例，然后通过服务容器获取服务名称为“view”的实例对象，即为视图创建工厂类 (`Illuminate\View\Factory`) 实例，接着通过视图创建工厂的 `make()` 方法来创建视图实例对象，其中参数为视图文件的名称，实际上是在视图模板文件路径中查找相应文件名的文件，视图文件将在后面进行创建，最后通过视图实例的 `with()` 方法添加数据，使其可以在视图文件中使用。

接下来进行视图模板文件的创建，首先在视图模板文件目录中创建“welcome.blade.php”文件，该视图组件规定模板文件要以“.blade.php”为后缀，文件名要与视图创建工厂的 `make()` 方法中的字符串参数相同。视图模板文件的具体内容如下：

文件 D:\WWW\lara\resources\views\welcome.blade.php

```
<h3> 在视图中显示学生信息: </h3>
学生 id:{{ $data['id'] }};<br />
学生 name:{{ $data['name'] }};<br />
学生 age:{{ $data['age'] }};<br />
```

在视图文件中可以按照 HTML 的格式创建视图，对于传入的数据可以通过“{{ 变量 }}”的方式输出。通过浏览器访问地址“http://127.0.0.1:10000/welcome”时，将输出视图，如图 1.10 所示。

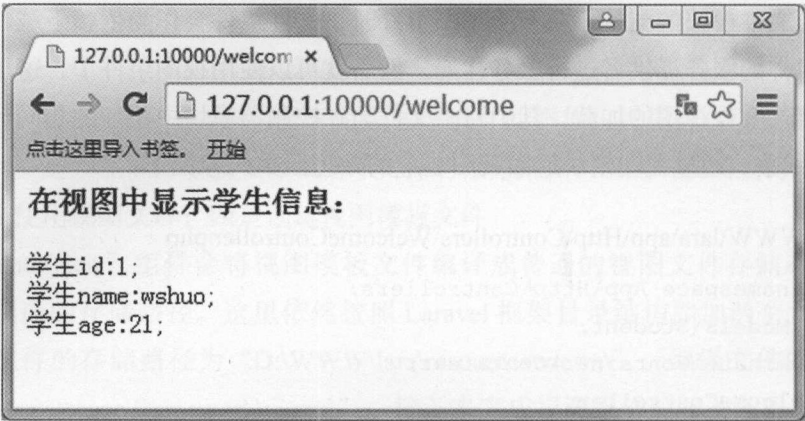


图 1.10 视图模块响应

至此，通过简单的几个步骤，一个功能比较全面的服务器端程序框架已经创建成功了，该框架不仅符合 MVC 的架构模式，而且通过服务容器来实现依赖注入，使得各组件间耦合度非常低，还有功能如此强大的模型类来操作数据库，这就是组件化开发的强大之处。该框架已经涵盖了 Laravel 框架的主要部分，相当于 Laravel 框架的骨骼，在后面的章节中将详细介绍这些模块的实现细节，逐渐揭开 Laravel 框架的面纱。

在本章中，首先介绍了组件化开发思想，然后介绍了 PHP 编程语言组件化开发的规范（PSR 规范）和工具（composer 工具），最后通过手动构建一个服务器端程序框架，将组件化开发思想、PSR 规范和 composer 工具的使用进一步深化。如果通过本章，读者能够理解组件化的开发思想和实现过程，那么本章的目的就达到了。在后续章节中，随着对 Laravel 框架了解的深入，再不时地回头看一下这部分内容，那么这种软件的构建思想将会内化为你思想的一部分，帮助你创造出更多优秀的产品。

第2章

Laravel 框架安装与调试环境建立

对于 Laravel 框架的学习，需要搭建相应的环境，这里介绍两种环境的搭建，分别是 Windows 环境和 Linux 环境。使用 Windows 环境的目的是方便大家学习，而搭建 Linux 环境的目的是方便大家部署。其中，Windows 环境采用 Windows 7 系统、SublimeText 编辑器、phpStudy 和 PhpStorm 集成开发环境，而 Linux 环境采用的是 Ubuntu 系统的 Server 版、LAMP 开发环境和 vim 编辑器。下面分别进行介绍。

2.1 Windows 开发环境搭建和调试

Windows 系统开发环境主要用于平时学习过程中代码的编写和调试，毕竟国内的大部分读者更加熟悉 Windows 系统。首先介绍一下需要用到的几款应用软件，SublimeText 作为代码编辑器不仅简单易用，而且有优美的界面风格和大量的应用插件，使代码编辑工作变得舒适而快捷。phpStudy 作为服务器软件集成环境支持多种版本组合，包括 Apache、Nginx、IIS 和 Lighttpd 四种服务器和 PHP(5.2~5.6) 的所有版本，版本切换非常方便，可以通过菜单命令打开、修改环境的配置文件，以及扩展和管理数据库等功能，使得在 Laravel 框架运行环境配置方面简便很多。PhpStorm 作为重量级的 PHP 集成开发环境不仅具有优美的编辑界面，而且还有动态调试功能，这对于学习 Laravel 这种大型框架来说是必不可少的。下面将分别安装相应的软件并进行配置，软件安装比较简单，将省略大部分步骤，主要介绍软件的相关配置。

2.1.1 Laravel 框架安装

在 Windows 环境中安装 Laravel 框架相对来说比较简单，首先需要安装 composer 包管理工具，下载的官方网址是 www.gocomposer.org，下载 Composer-Setup.exe 安装程序并进行安装，安装过程和注意事项可以参看第 1 章中的内容。

安装并完成环境变量配置后，可以在网站的根目录下（如 D:\WWW）单击鼠标右键，选择 use composer here 选项，并在命令行中输入命令 “composer create-project laravel/laravel --prefer-dist” 实现 Laravel 框架的安装。有些情况下无法下载，是因为被国内防火墙拦截了，可以通过网址 <http://www.golaravel.com/download/> 下载一键安装包或者用其他方式来下载安装。下载完成后，会在当前目录下生成一个 laravel 文件夹，该文件夹下就是整个 Laravel 框架项目了。

2.1.2 开发调试环境搭建

1. phpStudy 集成环境安装与配置

登录网址 <http://www.phpStudy.net/> 下载 phpStudy 集成环境，下载后得到安装程序，名称为 phpStudy_2014_setup.1413444920.exe，双击后一直单击 “next” 按钮即可安装成功，这里就不过多介绍了。安装成功后，双击 phpStudy，运行后的界面如图 2.1 所示。在默认情况下，phpStudy 集成环境默认的网站根目录是 “D:\WWW”，需要将下载并安装的 Laravel 框架复制到该目录下，也可以直接在该目录下通过 composer 工具下载 Laravel 框架。

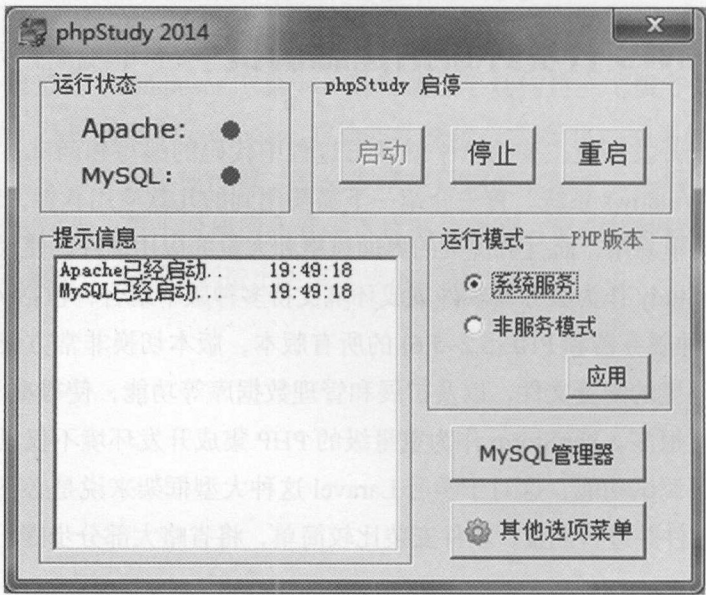


图 2.1 phpStudy 运行界面

Laravel 框架 5.1 版本要求 PHP 的版本不低于 5.5.9，可以在框架的 composer.json 文件中查找到 PHP 环境的需求配置。所以，需要将 phpStudy 集成环境的 PHP 版本设置为 5.6 版本，单击 “其他选项菜单” 后选择 “PHP 版本切换” 选项，打开 PHP 版本切换界面，如图 2.2 所示，这里选择 “Apache+PHP5.6n” 的组合后单击 “应用” 按钮。

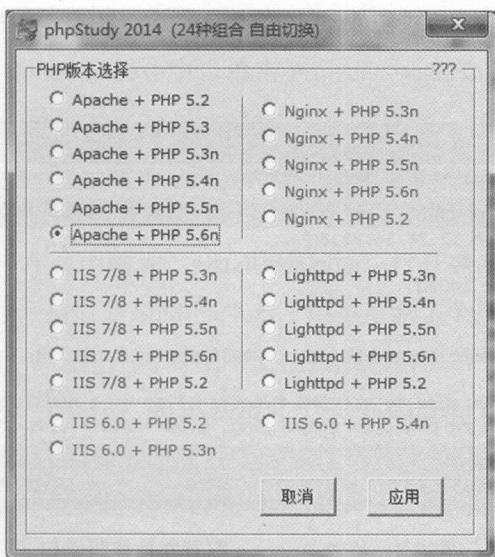


图 2.2 PHP 版本切换界面

下面进行站点域名管理配置。单击 phpStudy 运行界面中的“MySQL 管理器”按钮，选择“站点域名管理”选项，打开“站点域名设置”对话框，如图 2.3 所示，其中网站域名设置为“127.0.0.1:1000”，网站目录通过浏览选择 Laravel 框架下的 public 文件夹，第二域名可以不填写，网站端口填写 10000，也可以填写其他大的端口号，主要避开 80 和 8080 端口，防止端口冲突而无法使用 Apache 服务器。填写完毕后单击“新增”按钮保存设置。选择保存的设置并生成配置文件使配置生效。

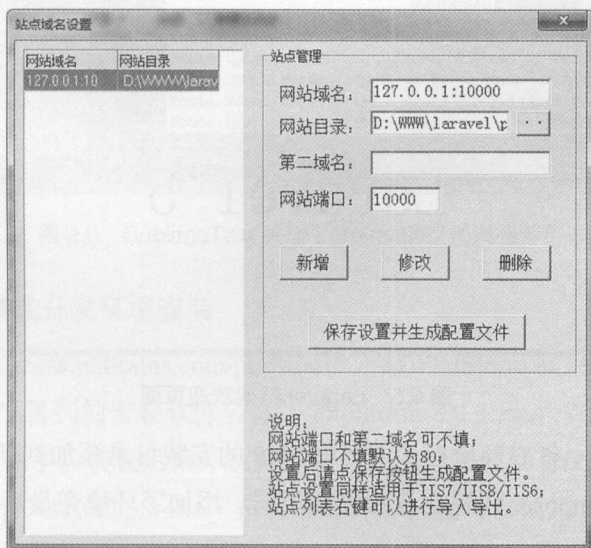


图 2.3 “站点域名设置”对话框

在完成站点域名配置后还需要修改 Apache 的配置文件来监听设置的端口（10000 端口），先单击启动界面中的“其他选项菜单”，在菜单中选择“打开配置”选项，选择 httpd-conf 文件，

通过文本编辑器打开后，在约 60 行处添加“Listen 10000”，用于添加监听的端口，修改文件内容如图 2.4 所示。

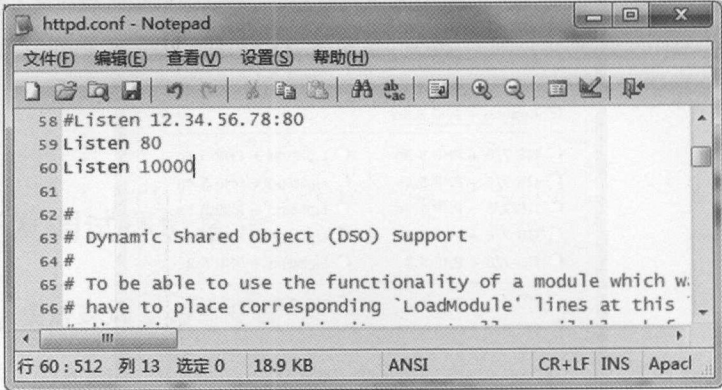


图 2.4 修改 Apache 配置文件监听端口

在修改完 Apache 配置文件后，由于 Laravel 框架需要 PHP 的 openssl 扩展支持，所以需要添加该扩展，在 phpStudy 运行界面中单击“其他选项菜单”，在菜单中选择“PHP 扩展及设置”、“PHP 扩展”，在其中选择 php_openssl，勾选的为配置的扩展，选中后重新启动。至此，满足 Laravel 框架运行的 phpStudy 集成环境基本配置就设置完了，通过浏览器访问地址 <http://127.0.0.1:10000/> 即可查看到 Laravel 框架欢迎页面，如图 2.5 所示，说明 phpStudy 环境配置成功了。

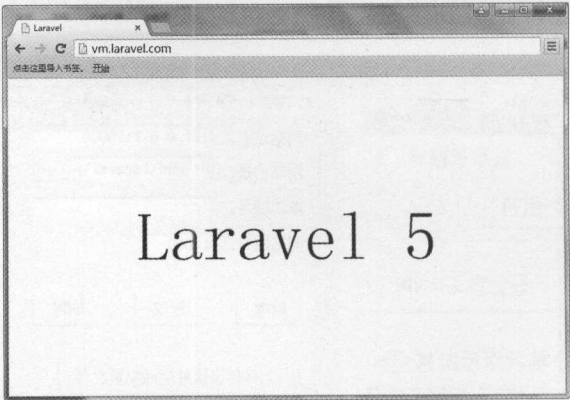


图 2.5 Laravel 框架欢迎页面

在安装完 phpStudy 集成环境后，需要将 PHP 的安装目录添加到环境变量中，添加方法与上一小节中添加 composer 环境变量的方法相同。添加了环境变量后就可以在 cmd 命令窗口中直接使用 PHP 命令了。

2. SublimeText 编辑器安装与配置

登录官网 <http://www.sublimetext.com/> 下载 SublimeText 安装程序，目前正式版本为 SublimeText 2，而 SublimeText 3 当前为 beta 版，正处于公共测试阶段，可以直接下载

SublimeText3 使用，下载后一直单击“next”按钮安装即可。安装完成后，首先将下载的 Laravel 框架项目添加到 SublimeText 中，单击 Project → add folder to project 并选择 laravel 文件夹即可添加。接下来需要配置 SublimeText 的主题界面，在 Preferences 选项中可以通过字体设置改变字体大小，通过主题方案选择不同的主题，通过这两个选项基本上可以打造一个适合自己风格的主题界面。接着安装几个重要的 SublimeText 插件，通过按“Ctrl+Shift+P”快捷键打开插件管理器，在其中输入“install package”打开 Package Control:Install Package 包管理器。如果无法找到，是因为 SublimeText 3 版本更新了 Python 函数库，导致很多基于 Python 开发的插件不能正常工作，可以在网上找到多种解决方法，这里就不赘述了。调出包管理器后，在其中输入“laravel”可以列出几个关于 Laravel 的插件，可以将其中列出的 Laravel 5 插件安装上。这时在 Laravel 项目中添加代码就支持代码提示了，代码提示界面如图 2.6 所示。至此，Laravel 框架的基本配置就完成了，如果需要其他功能可以继续添加相应的插件。

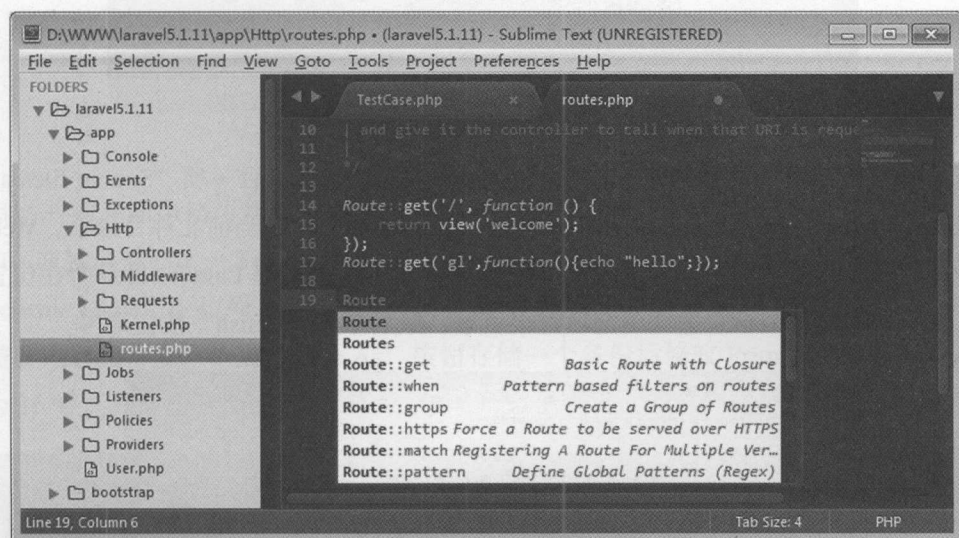


图 2.6 SublimeText 关于 Laravel 框架代码提示界面

3. PhpStorm 集成开发环境安装

登录官网 <https://www.jetbrains.com/phpstorm/> 下载 PhpStorm 集成开发环境，这里下载的是 8.0.3 版本，下载后得到的安装软件名称为 PhpStorm-8.0.3.exe，安装也非常简单，一直单击“next”按钮就可以了。安装后第一次打开会提醒用户加载已有的配置文件。之后会提醒注册，可以通过单击“Buy PhpStorm”按钮购买许可，也可以试用 30 天，这里选择的是试用 30 天，确定后就可以启动 PhpStorm 了。第一次启动后，会提示用户选择初始化配置，包括主题设置、编辑器字体设置等，可以在此时进行设置，也可以使用默认设置，然后在开发窗口界面进行修改，这里选择默认设置。这时出现 PhpStorm 项目创建界面，如图 2.7 所示，可以创建、打开新项目，由于使用已有的 Laravel 框架创建项目，所以选择第三项“Create

New Project from Existing Files”，即从已经存在的文件创建新项目。

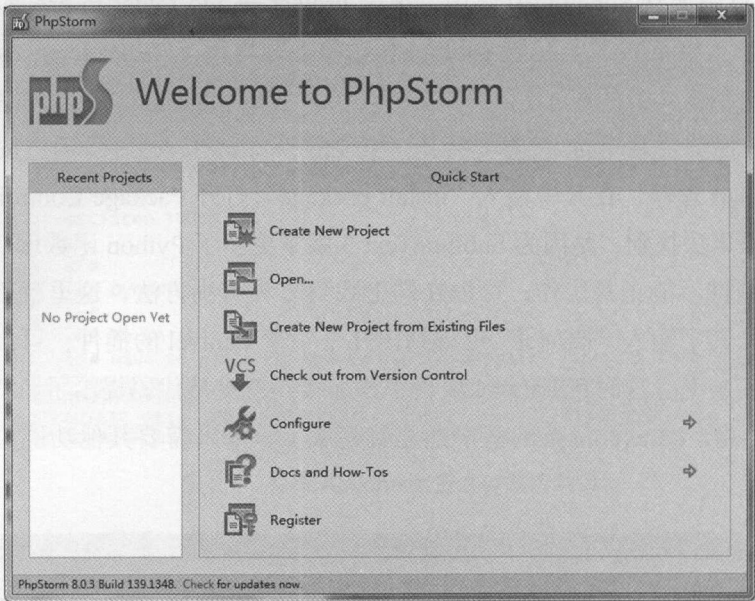


图 2.7 PhpStorm 项目创建界面

接着出现项目类型选择界面，如图 2.8 所示，这里选择最后一项“Source files are in a local directory, no Web server is yet configured”，即源文件在当前文件目录下，Web 服务器还没有配置，出现项目根目录选择界面，如图 2.9 所示，选择 Laravel 目录所在位置，并单击上面的“Project Root”选项确定项目根目录，最后单击“Finish”按钮即可。

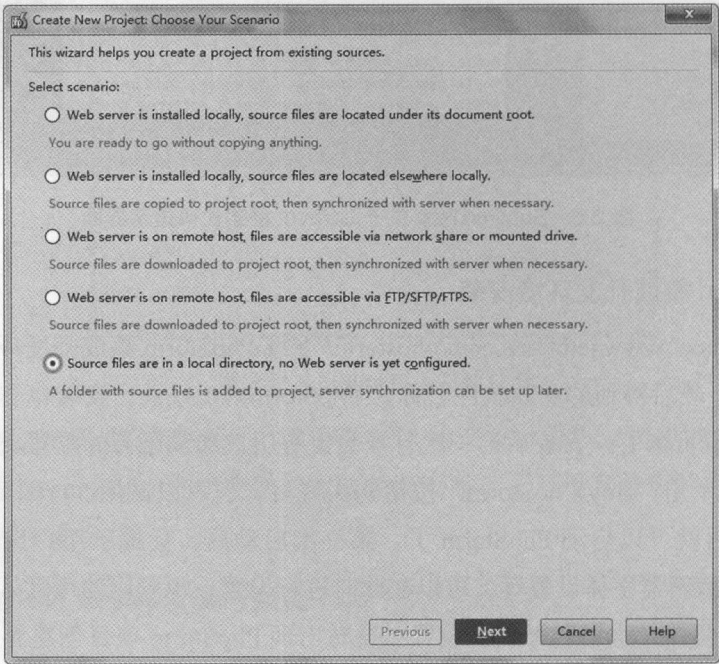


图 2.8 项目类型选择界面

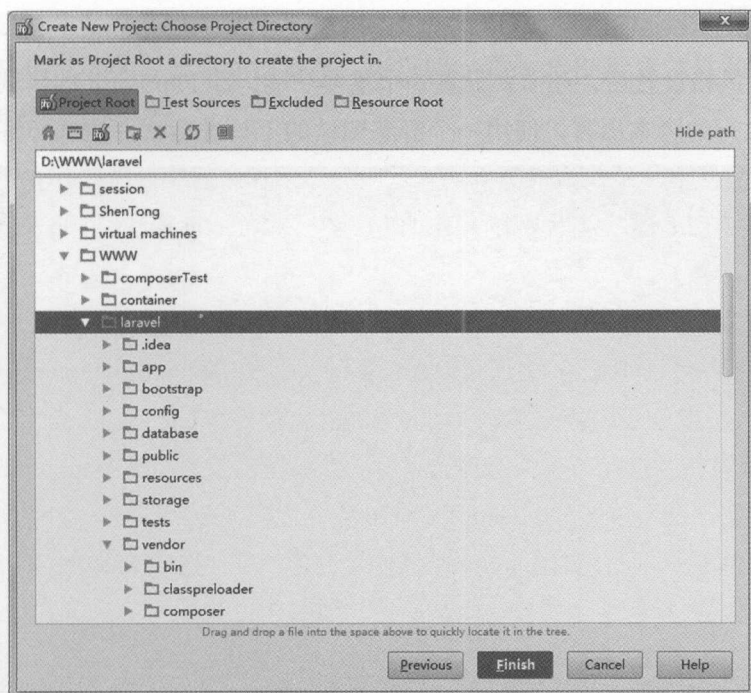


图 2.9 项目根目录选择界面

启动后的界面风格可能不是你想要的，主要是字体太小，主题不美观，可以进一步进行修改。通过选择 File → Settings → Editor → Colors&Fonts → Font，会打开 IDE 的主题配置，在“Scheme name”下拉列表框中选择 Monokai 主题，需要注意的是在修改主题的字体和字号前需要先存储备份，单击“Save As...”按钮存储一个备份后修改 Primary font 为 Consolas、Size 为 16 号，界面设置如图 2.10 所示，读者可以根据自己的喜好选择合适的主题风格。

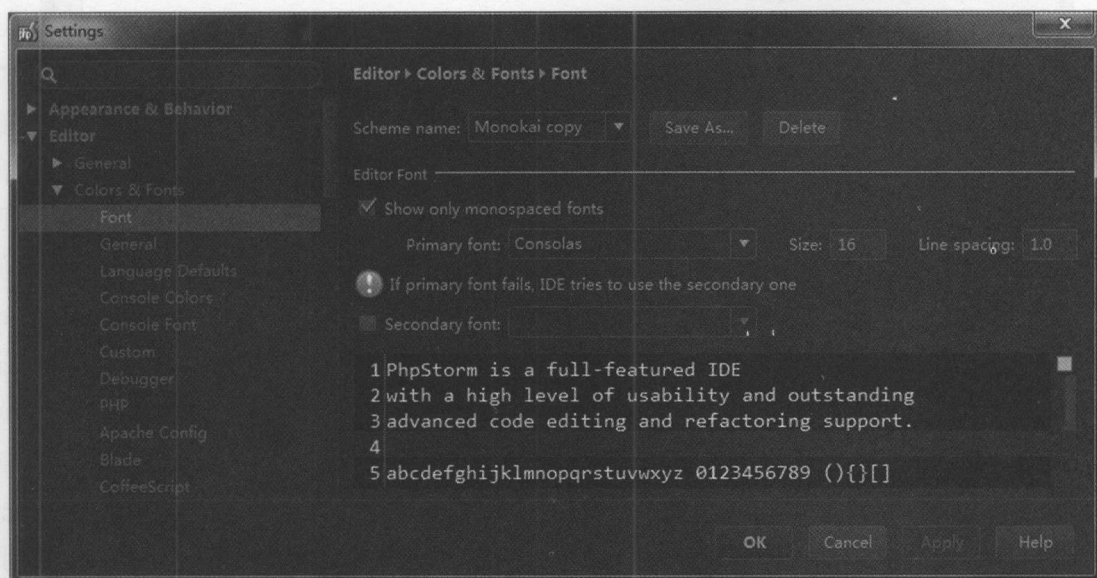


图 2.10 PhpStorm 主题风格设置界面

4. PhpStorm+Xdebug 调试环境搭建

在完成界面风格设置后，还需要设置一个重要功能，即 PhpStorm 的调试功能，对于如 PHP 这种动态执行的脚本语言，使用一个静态调试的工具进行软件调试必将事半功倍，这样就可以通过设置断点、单步执行等方式查看服务器端程序的执行流程并查看到实例对象的相关状态等。要实现上述功能需要在 PHP 环境中安装 Xdebug 模块，同时配置 PHP 环境和 PhpStorm 的相关参数。

如果是自己安装 PHP 环境并添加 Xdebug，则需要两个版本统一。对于 phpStudy 集成环境，在 PHP 安装目录下默认添加了 Xdebug 模块，如 D:\phpStudy\php56n\ext 目录下的 xdebug.dll 文件，所以这里只需要修改一下 PHP 的配置文件 php.ini 即可使用调试功能。打开 phpStudy 运行界面，选择“其他选项菜单”，打开配置文件，选择 php.ini 后通过编辑器打开，直接在该文件最后添加 “[xdebug]” 项和相关配置信息，添加配置后如图 2.11 所示。完成 phpStudy 配置后，通过选择 phpStudy 操作界面的“其他选项菜单”，查看 phpinfo 项，即可在 xdebug 项中查看到刚才配置的调试信息，如图 2.12 所示。

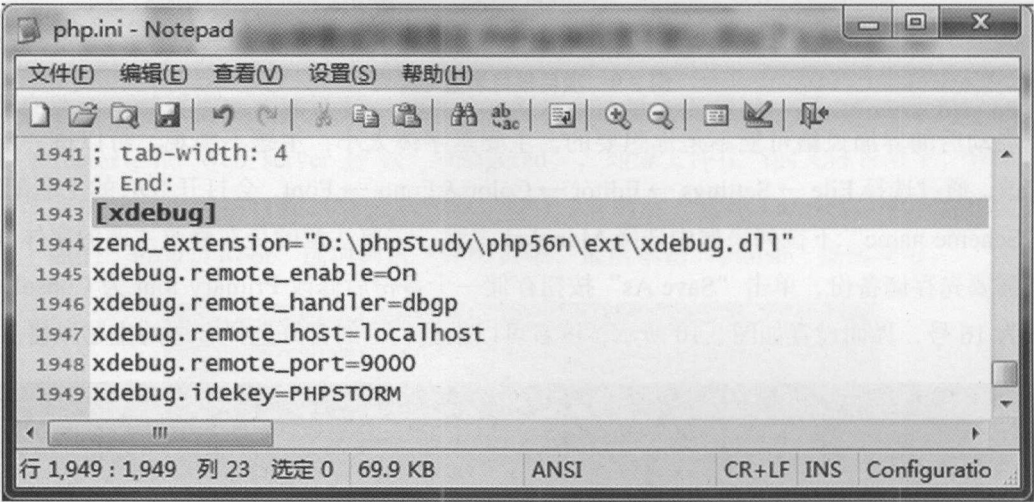


图 2.11 PHP 关于 xdebug 配置

xdebug.remote_enable	On	On
xdebug.remote_handler	dbgp	dbgp
xdebug.remote_host	localhost	localhost
xdebug.remote_log	no value	no value
xdebug.remote_mode	req	req
xdebug.remote_port	9000	9000

图 2.12 xdebug 配置信息

在完成 PHP 环境的 xdebug 配置后，接下来需要配置 PhpStorm 集成开发环境。首先设置 PhpStorm 的调试端口，选择 File → Settings，在搜索栏中输入“xdebug”，会搜索到

PHP 的 Debug 选项配置界面,如图 2.13 所示。然后在“Debug port”文本框中设置 9000,该值默认情况下即为 9000,如果端口冲突,可以同时修改 PHP 配置文件和 PhpStorm 设置中的端口号,两者要保持一致。

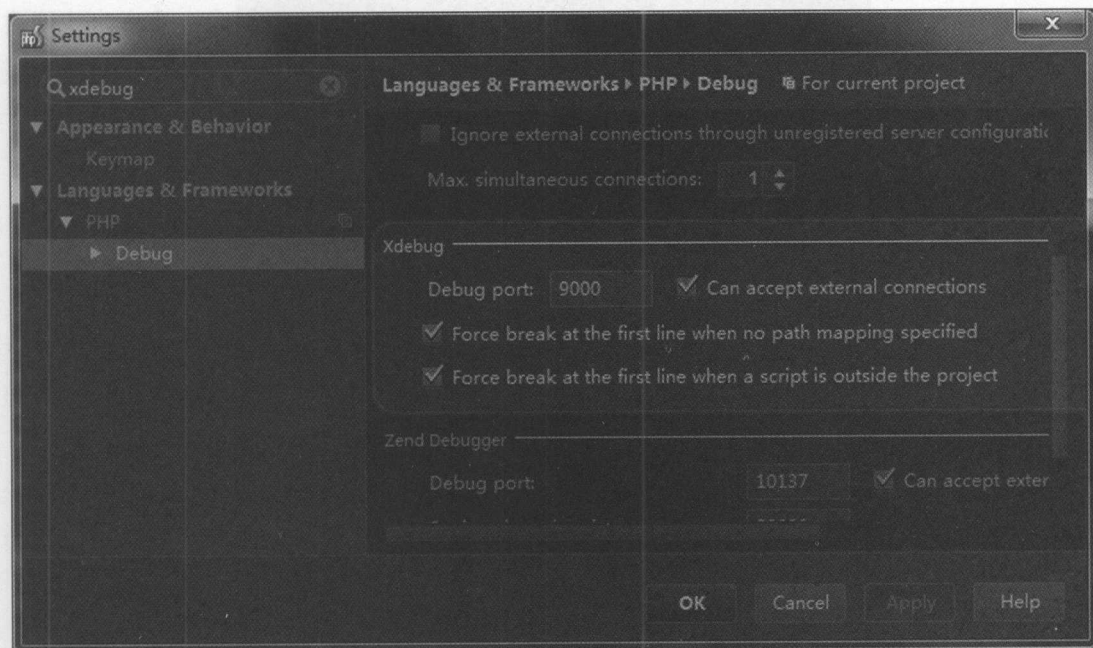


图 2.13 PhpStorm 关于 Debug 的选项配置界面

接下来配置调试站点信息。在 PhpStorm 的右上角有执行和调试的图标,如图 2.14 所示,在没有配置调试参数的情况下图标为灰色,单击左边的下拉按钮可以选择 Edit Configurations 来配置所要调试的 Web 程序,配置界面如图 2.15 所示。单击左上角的“+”按钮,选择 PHP Web Application,单击 Server 右侧的“...”按钮,弹出 Web 服务器访问的配置界面,如图 2.16 所示。在“Name”文本框中设置 Web 名称,这里设置为 laravel,将 Host 地址设置为 127.0.0.1,将 Port 设置为 10000,在 Debugger 下拉列表框中选择 Xdebug。配置完成后单击“OK”按钮返回上一级配置界面,在“Start URL”文本框中可以输入网站的 URL,这里设置的 URL 为“/”,即为访问网站的根目录。配置完成后单击“Apply”按钮进行应用,单击“OK”按钮关闭后运行和调试的图标就变成了绿色,这时可以在网站的入口文件(如 laravel/public/index.php)代码的左侧单击即可添加断点,单击调试图标(小虫子图标)即可进行调试,程序会执行到断点处停止。菜单栏的 run 菜单项下具有相应的调试指令,包括单步调试(F7)、过程调试(F8)、恢复运行直到下一断点(F9)等,通过这些指令可以完成 Web 程序的静态调试。

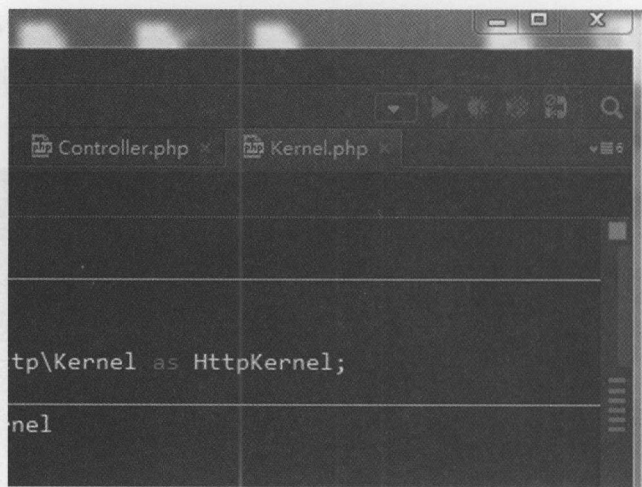


图 2.14 执行和调试的图标位置

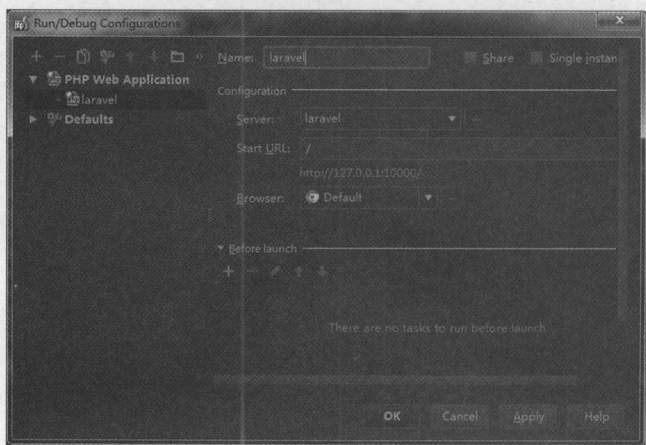


图 2.15 程序调试配置界面

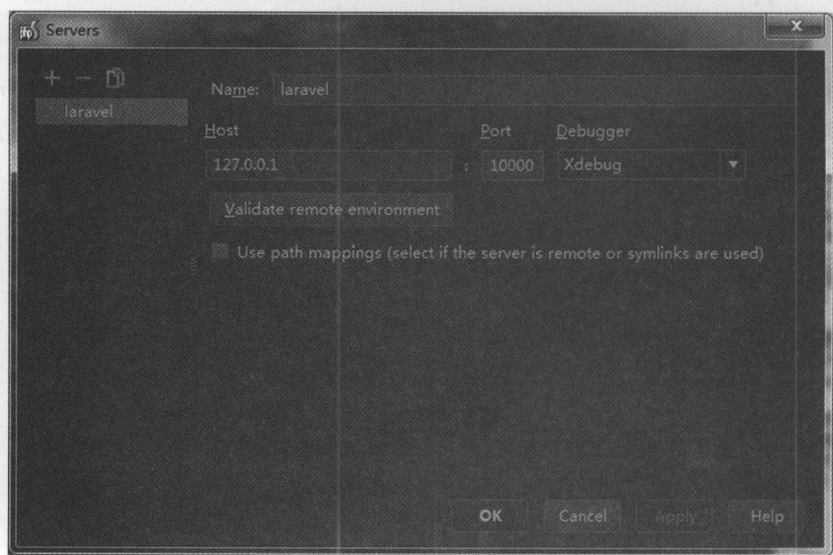


图 2.16 Web 服务器访问的配置界面

2.2 Linux 开发环境搭建

Linux 开发环境是在 vmware workstation 11 中搭建的，vmware 的安装比较简单，在网上也有很多资源，这里就不做介绍了，下面主要介绍 LAMP 环境搭建和 Laravel 框架安装。

2.2.1 LAMP 环境搭建

1. Linux 系统虚拟机搭建

首先，选择的 Linux 系统是 Server 版的 Ubuntu 系统，版本号为 14.04.3 LTS，LTS 表示长期支持版本，官方网址为 <http://www.ubuntu.com/download/server>，下载页面如图 2.17 所示。

Ubuntu 系统在 vmware 下的安装网上也有很多资料，这里不再过多叙述，只讲几处容易忽视的问题。在选择安装来源时选择“稍后安装操作系统”，这样用户可以调整配置的选项，如图 2.18 所示。

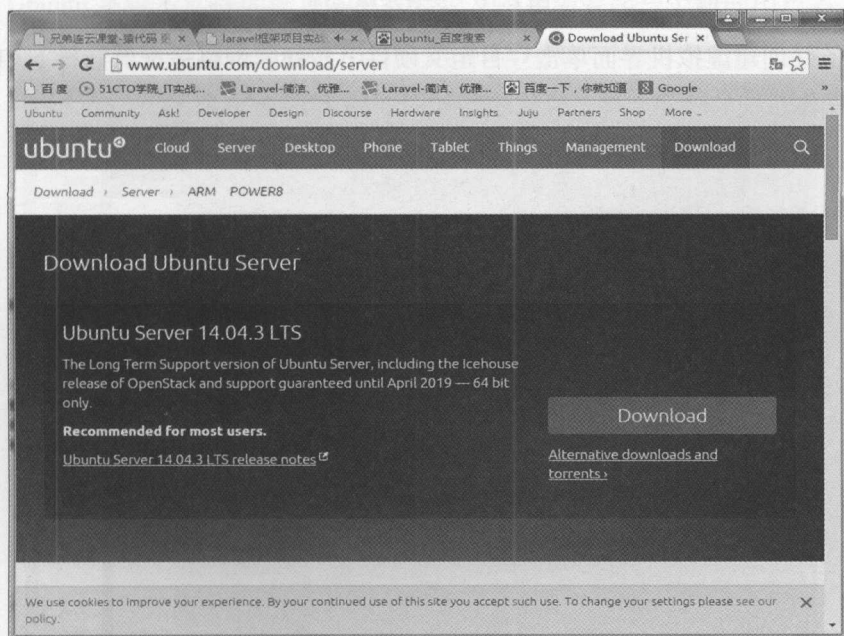


图 2.17 Server 版 Ubuntu 系统下载页面

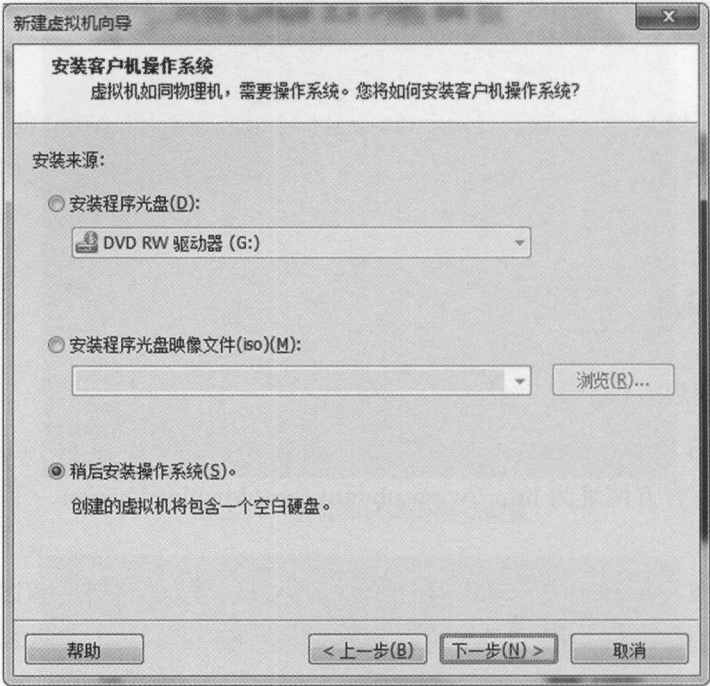


图 2.18 安装来源选择

在已准备好创建虚拟机界面单击“自定义硬件”按钮，网络适配器选择 NAT 模式，在光驱中加载操作系统的映像文件，操作步骤如图 2.19 和图 2.20 所示。

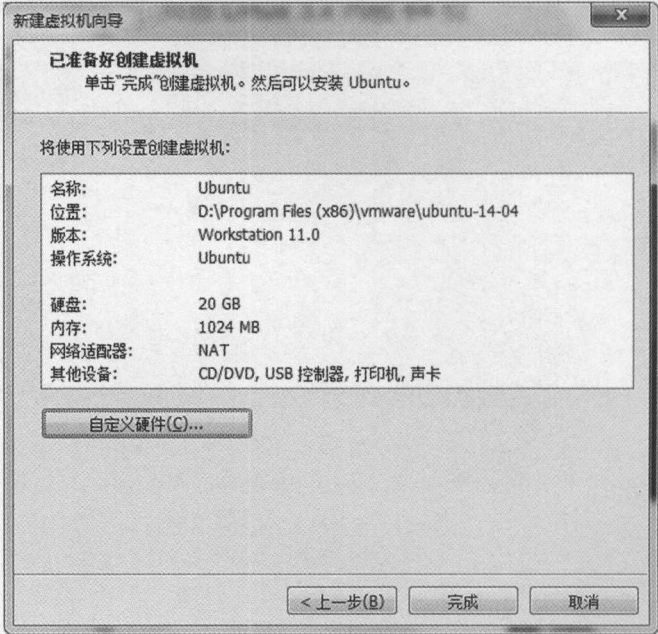


图 2.19 自定义硬件

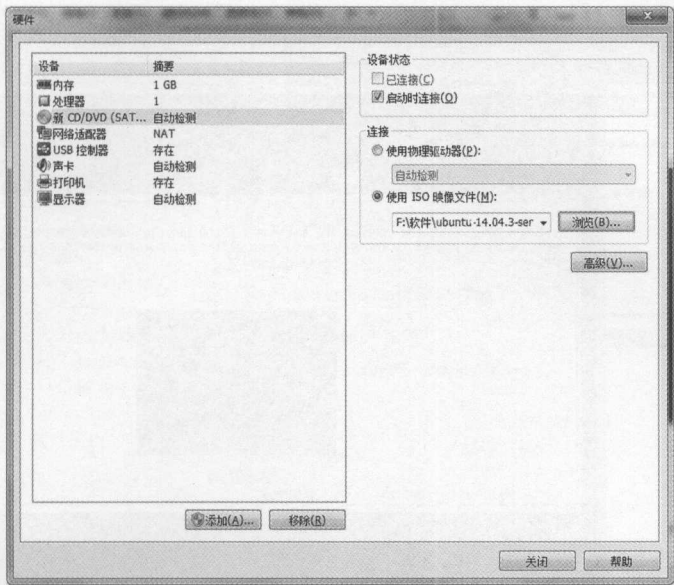


图 2.20 加载映像文件到光驱

下面开启此虚拟机进行 Ubuntu 系统的安装，这里也只介绍几个容易忽略的步骤。Server 版的 Ubuntu 系统不支持远程 root 用户登录，所以需要起一个新的用户名，通过这个用户名来登录，登录后可以切换到 root 用户，如图 2.21 所示。

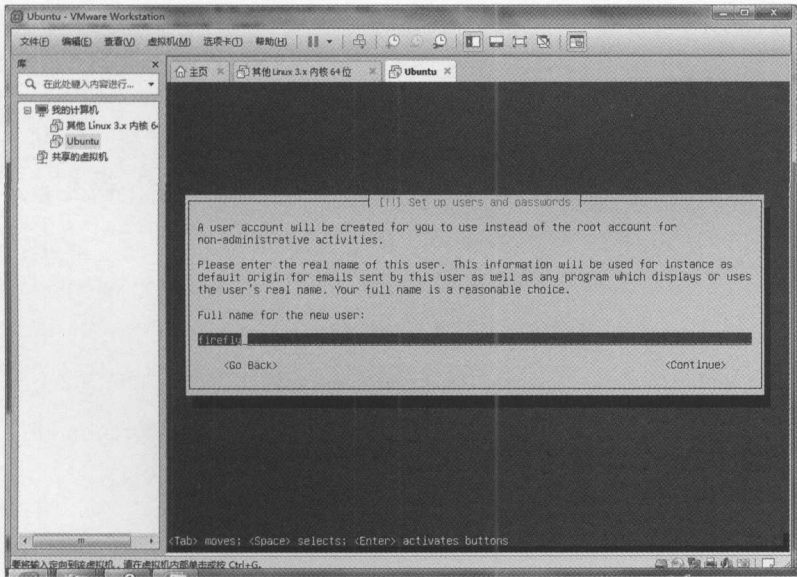


图 2.21 创建 Ubuntu 系统登录用户名和密码

在选择安装软件时，需要安装的软件通过按空格键来确认，前面括号中带星号的表示已被选中。这里选择安装 OpenSSH，用来后面的远程连接，其他的不安装，这里虽然也包含 LAMP 环境的自动安装，但是可能与 Laravel 需求版本不匹配，所以手动安装 LAMP 环境，如图 2.22 所示。

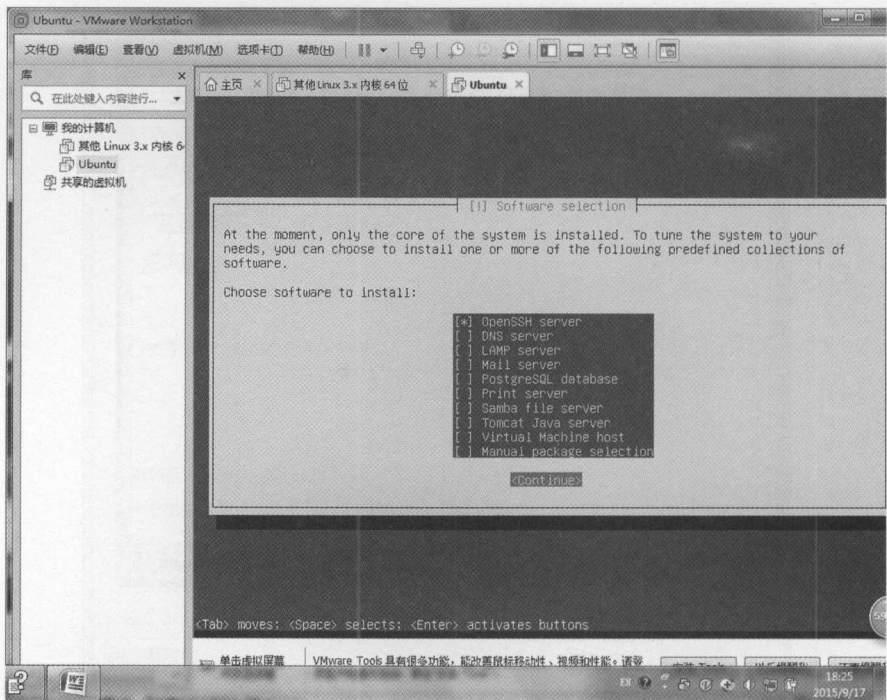


图 2.22 选择安装软件

在安装虚拟机时，由于网卡选择的是 NAT 模式，所以 Ubuntu 系统安装完成后可以直接联网（在本机可以联网的前提下）。

2. putty 连接工具

虚拟机的 Linux 环境操作起来不够方便，也不支持命令复制、粘贴等操作，所以用 ssh 工具通过本地计算机连接虚拟机中的 Linux 系统。这里使用的是 putty，其官方下载地址为 <http://www.putty.nl/download.html>。putty 是一个绿色软件，下载后可以直接使用。开启后在 Host Name 文本框中输入 IP 地址，可以在 Saved Sessions 中起个名字并保存，以备下次使用。putty 本身的界面不是很美观，字体小而且配色不舒服，需要进行设置，设置项在刚启动时的左侧栏。在 Window → Appearance 中，单击 change 按钮，选择 Consolas 字体、16 号大小。在 Window → Colours 中，将 Default Foreground 设置为暗绿色，即 RGB 为 0、130、0，将 Default Background 设置为纯黑色，即 RGB 为 0、0、0，配置选项及最终效果如图 2.23 和图 2.24 所示。

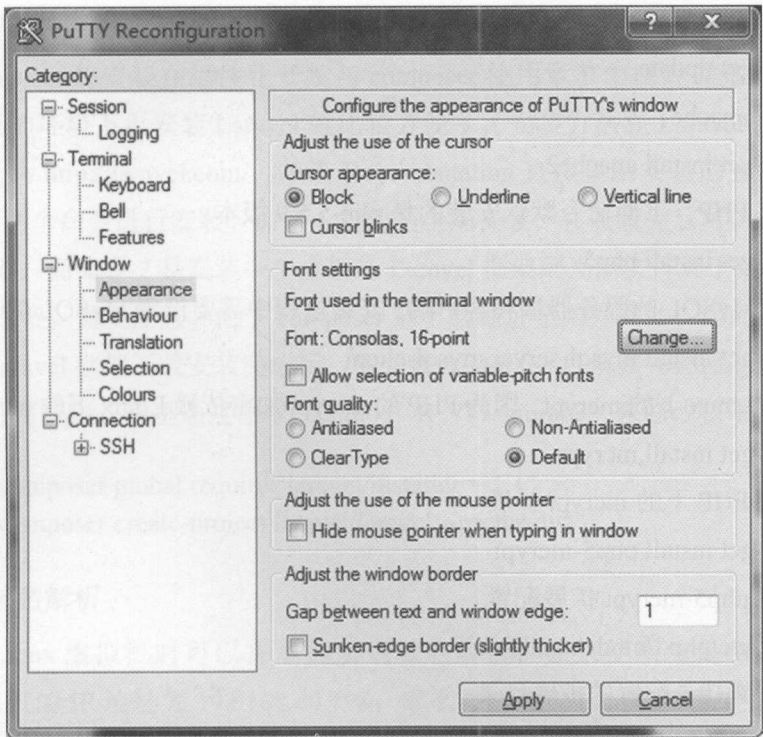


图 2.23 字体配置选项

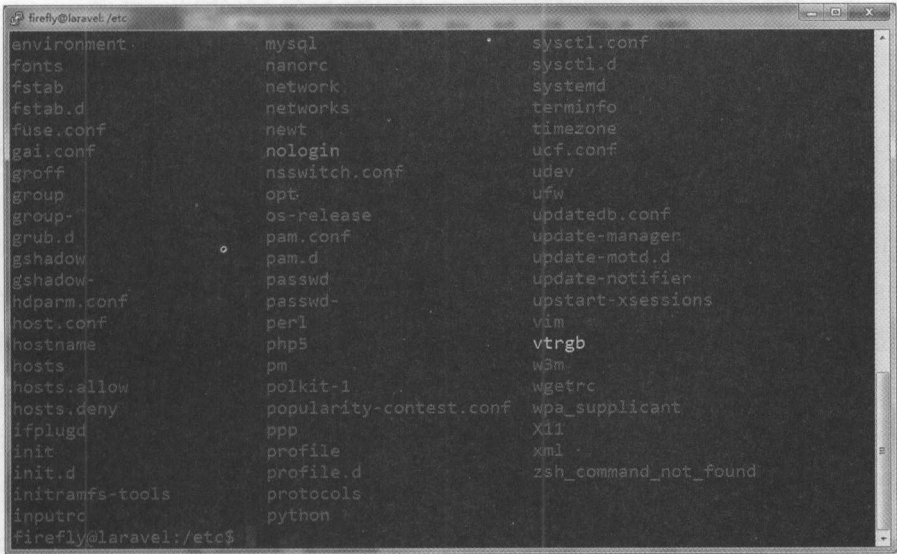


图 2.24 配色效果

3. Apache、Mysql 和 PHP 的安装

启动操作系统后，通过如下命令完成 Apache、MySQL 和 PHP 等相关软件的安装和配置。

(1) 切换到 root 用户。

命令：sudo su

(2) 更新 apt-get。

命令：apt-get update

(3) 安装 apache。

命令：apt-get install apache2

(4) 安装 PHP，下面命令默认安装的是 php-5.5.9 版本。

命令：apt-get install php5

(5) 安装 MySQL 的服务器端和客户端，安装过程中需要设置 MySQL 的 root 用户密码。

命令：apt-get install mysql-server mysql-client

(6) 安装 Linux 下的 mcrypt，因为 PHP 的 mcrypt 加密依赖 Linux 下的 mcrypt 加密扩展。

命令：apt-get install mcrypt

(7) 安装 PHP 下的 mcrypt 扩展。

命令：apt-get install php5-mcrypt

(8) 查看 php5-mcrypt 扩展配置。

命令：ls /etc/php5/mods-available

(9) 进入 PHP 5 配置的扩展目录。

命令：cd /etc/php5/apache2/conf.d

(10) 配置 mcrypt，即将 mcrypt 扩展配置文件设置软链接到当前目录下。

命令：ln -s /etc/php5/mods-available/mcrypt.ini /etc/php5/apache2/conf.d/

(11) 测试安装环境，进入到 apache 服务器的默认根目录。

命令：cd /var/www/html/

(12) 创建一个 phpinfo.php 文件，文件内执行 phpinfo() 函数，通过 ifconfig 查询新建的 Linux 服务器的 IP 地址并通过浏览器访问。

这里需要懂一点 Linux 系统的命令行操作，若不了解可以查阅 Linux 方面的书籍，网上也有很多 LAMP 环境的搭建资料可以查阅。在访问的网页中查看是否存在 pdo_mysql、mcrypt 等扩展，如果存在，说明我们需要的 LAMP 环境已经搭建完毕，接下来搭建 Laravel 环境。

2.2.2 Laravel 安装

Laravel 框架主要的特点是支持组件化开发，也就是说对于 Laravel 不同的组件，可以随时增加或删减，不会对其他组件产生影响。与 Laravel 框架对应的就是全功能框架，这种框架用户或者全部使用，或者不使用，如果使用其部分功能则需要重构，比如国内使用比较多的 ThinkPHP 框架。因此，组件化开发的框架要更加灵活。组件化开发需要有命名空间的支持，所以在 Java、Ruby 等语言中很常见，但是在 PHP 开发的前期是不支持组件化开发的，因为 PHP 在 5.3 版本以后才引入了命名空间。

1. 通过 composer 安装 Laravel 框架

对于 composer 的安装在组件化开发与 composer 使用章节中已经详细介绍过了，在有 composer 工具的环境下再安装 Laravel 就比较方便了，安装方法在 Laravel 官网上给出了详细介绍，网址为 <http://laravel.com/>，单击 Documentation 直接显示安装的方法。在 Linux 下可以通过下面两个命令进行安装，命令 1 指的是全局安装，在安装完后可以通过命令新建多个 Laravel 框架，而命令 2 是安装一个全新的 Laravel 框架到当前文件夹下。这里使用命令 2 安装 Laravel 框架。需要注意，由于 composer 的下载地址是国外网站，因此可能无法正常下载安装。在 Laravel 框架下载安装完成后，需要修改 storage 目录和 /bootstrap/cache 目录的权限为 apache 可写权限，这里通过命令“chmod -R 777 storage”来修改权限。

命令 1: `composer global require "laravel/installer=~1.1"`

命令 2: `composer create-project laravel/laravel --prefer-dist`

2. 本地域名解析

在访问 Linux 虚拟机时可以在本地实现域名解析，如果使用的是 Windows 7 系统，而 Linux 虚拟机的 IP 地址为 192.168.40.128，那么可以在 Windows 7 系统的“C:\Windows\System32\drivers\etc”目录下找到 hosts 文件，打开记事本，添加一行“192.168.40.128 vm.laravel.com”，保存后当再次访问 Linux 虚拟机的服务器时，直接在浏览器中输入 vm.laravel.com 就可以访问了。因为在安装 apache 服务器时，默认的 Web 根目录为“/var/www/”，在此目录下默认有一个 index.html 文件，所以主机访问 vm.laravel.com 地址时，访问的就是这个 apache 的 index.html 文件。

下面修改 Linux 服务器的 apache 虚拟主机配置，通过命令“vim /etc/apache/apache2.conf”来修改 apache 的配置文件，在这个配置文件中可以看到一个“<Directory>”配置项，默认该配置项设置的是“/var/www/”目录，如果安装的 Laravel 框架在这个目录之下，这里就不需要修改，如果不在这个目录下，那么就需要将这个配置项再复制一份，添加到 Laravel 框架的目录上，当然也可以直接修改。

下面来配置站点信息，需要进入到“/etc/apache2/sites-available”目录下，在该目录下可以看到一个配置文件“000-default.conf”，这个文件就是配置 apache 默认的 index.html 站点的文件，可以按照它的配置来设置 Laravel 站点。这里通过命令“cp 000-default.conf Laravel.conf”复制一份，并修改相应的两条配置（一个是域名，另一个是文件路径）：

```
ServerName vm.laravel.com
```

```
DocumentRoot /var/www/laravel/public/
```

接着创建该配置文件的软链接，即通过命令“ln -s /etc/apache2/sites-available/laravel.conf ../sites-enabled/”将刚才创建的 laravel.conf 配置文件软链接到“../sites-enabled/”目录下。此时，通过“apachectl restart”命令重启 apache 就可以使配置生效。

经过以上步骤，Laravel 框架在 Linux 虚拟机中的安装就完成了，可以通过本地计算机的浏览器访问 Laravel 框架程序，如果出现欢迎页面，则表示 Laravel 框架在 Linux 系统中运行正常。

3. 优雅访问路径设置

在 apache 默认设置下，访问其他路由形式如“vm.laravel.com/index.php/test”，如果想修改为 RESTful API 的形式（vm.laravel.com/test），需要启动 apache 的 rewrite 重定向配置，该配置文件已经在目录“/etc/apache2/mods-available/”下存在，需要做的是通过命令“ln -s/etc/apache2/mods-available/rewrite.load/etc/apache2/mods-enabled/”创建一个软链接。接下来还需要查看目录权限配置，该配置在文件“/etc/apache2/apache2.conf”中，查看“/var/www/”目录的 AllowOverride 权限，默认情况下为“None”，需要修改为 all。之所以可以实现重定向，是因为在“/laravel/public/”目录下有一个 .htaccess 隐藏文件，该文件负责 Web 目录下的网页配置，可以实现网站重定向、自定义 404 错误页面、改变文件扩展名等功能。前面做的相关配置就是使这个文件起作用。

本章概要地介绍了 Laravel 框架运行环境的搭建，这部分内容很容易在网上找到，所以并没有进行很详细的介绍，由于软件版本不同，如果在安装过程中遇到问题，可以在网上查找相关资料。对于学习 Laravel 框架，调试环境还是必须的，因为可以像调试编译语言程序的代码一样一步步执行，并能够了解每一步程序都做了什么、所有的变量如何变化。建议读者学习时采用 PhpStorm+Xdebug 的调试方式进行学习。

第3章

Laravel 框架中常用的 PHP 语法

Laravel 框架之所以被称为优雅的代码并不是因为它发明了很多新的东西，而是将已有的技术发挥到了一定的高度，而这些技术包括各个方面，如程序的架构思想、设计模式的运用及 PHP 新语法的使用。新事物的产生一定具有它得天独厚的优势，对于一种语言的新语法也是一样，新的语法一定能够解决某一类棘手的问题。正因为在 Laravel 框架中使用大量的 PHP 新语法，包括命名空间、匿名函数、反射机制、后期静态绑定等，才使得 Laravel 框架显得简洁而易扩展。正因为该框架应用了大量的 PHP 新语法，所以需要对此部分内容有一定的了解，才能理解 Laravel 框架实现的方式，并能在以后的工作中使用这些新技术提高自己的生产力。

3.1 组件化开发语法条件

3.1.1 命名空间

命名空间最初的设计是为解决命名冲突而产生的一种包装类、函数和常量的方法，很多早些年书中对命名空间的介绍也只限于此。通过对组件化开发和 composer 使用的了解，应该可以看到命名空间另一个重要的作用是为组件化开发提供了可能，也就是通过命名空间来组织文件，使得某个组件的文件的路径和命名空间具有一定的关系，最终可以直接通过命名空间找到相应的文件。正因为这种特性，使得 composer 管理工具可以方便地管理组件包的文件关系，再通过 PSR 规范将不同人开发的资源包方便地组合在一起。下面介绍有关命名空间及文件加载的相关内容。

1. 命名空间的定义

命名空间是通过关键字 `namespace` 来定义的，如果一段 PHP 代码要通过命名空间来封装，则命名空间的声明需要在这部分代码之前。具体实例如下所示：

```
<?php
namespace App\Http;
```

```
use Illuminate\Foundation\Http\Kernel as HttpKernel;
class Kernel extends HttpKernel
{
    // 其他代码内容
}
```

上述例子中定义了一个命名空间 `App\Http`，并在命名空间下定义了 `Kernel` 类，因此 `Kernel` 类的完整名称应该为 `App\Http\Kernel`。本质上，`Kernel` 类文件的文件路径与命名空间是相互独立的，即在任何目录下都可以定义 `App\Http` 命名空间，但为了命名的规范及后期文件包含的方便，一般将命名空间与文件路径定义为相同的名称，而文件名和类名定义为相同的名称，这样通过一个类的完整名称，就可以确定这个类所在文件相对于根目录的位置，为后期文件包含提供便利，这也是 PSR 规范规定的部分内容。

PHP 支持两种获取和使用当前命名空间的方法，分别是 `__NAMESPACE__` 魔术常量和 `namespace` 关键字。通过魔术常量 `__NAMESPACE__` 可以直接获取当前命名空间名称的字符串。如果是全局的代码，即不包括在任何命名空间中的代码，通过该魔术常量将获取一个空的字符串。关键字 `namespace` 用来显式访问当前命名空间。需要注意的是，如果没有定义命名空间，即为全局空间，相当于根空间，通过 “\” 来表示。

命名空间 `App\Http`:

```
<?php
namespace App\Http ;
echo __NAMESPACE__ ;
输出 : " App\Http "
```

全局代码 :

```
<?php
echo __NAMESPACE__ ;
输出 : ""
```

通过 `namespace` 显示访问:

```
<?php
namespace App\Http ;
class Kernel
{
}
$a = new namespace\ Kernel(); // 即 new App\Http\Kernel();
```

2. 命名空间的使用

命名空间的使用是通过 `use` 关键字来实现的。PHP 命名空间的使用方式与文件系统的使用方式有些相似。在文件系统中，通常访问一个文件有三种方式。

方式一：通过相对路径访问文件，如 `file.txt`，则它会被解析为当前路径加文件名的形式，即 `currentdirectory/file.txt`，`currentdirectory` 可能是很多级文件夹的目录。

方式二：通过带限定的相对路径访问文件，如 `subdirectory/file.txt`，则它会被解析为当

前路径加带限定的相对路径文件名形式，于是访问的文件为 `currentdirectory/sub-directory/file.txt`。

方式三：通过绝对路径访问文件，如 `C:/currentdirectory/file.txt`。

PHP 命名空间的使用也是同样的道理，通常有三种形式。

方式一：非限定名称或不包含前缀的类名称，如 `$a=new Kernel()`。如果当前命名空间是 `currentnamespace`，`Kernel` 将被解析为 `currentnamespace\Kernel`。如果当前空间是根空间，即是全局的，则 `Kernel` 会被解析为 `\Kernel`。

方式二：限定名称或包含前缀的名称，例如 `$a = new subnamespace\ Kernel ()`。如果当前的命名空间是 `currentnamespace`，则 `Kernel` 会被解析为 `currentnamespace\subnamespace\Kernel`。如果当前空间是根空间，即是全局的，`Kernel` 会被解析为 `subnamespace\Kernel`。

方式三：完全限定名称或包含了全局前缀操作符的名称，例如，`$a = new \currentnamespace\ Kernel ()`。在这种情况下，`Kernel` 总是被解析为 `\currentnamespace\Kernel`。

下面通过实例来介绍三种方式使用的区别。

文件 `file1.php`:

```
<?php
namespace App\Http\Controllers\Auth ;
function index ()
{
    echo '命名空间 '.__NAMESPACE__."<br>";
}
class Controller
{
    public static function index()
    {
        echo '命名空间 '.__NAMESPACE__."<br>";
    }
}
```

文件 `file2.php`:

```
<?php
namespace App\Http\Controllers ;
include 'file1.php';
function index ()
{
    echo '命名空间 '.__NAMESPACE__."<br>";
}
class Controller
{
    }
```

```

static function index()
{
    echo '命名空间 ' . __NAMESPACE__ . '<br>';
}
}

/* 非限定名称 */
// 解析为函数 \App\Http\Controllers\index()
index ();
// 解析为类 \App\Http\Controllers\Controller 的静态方法 index()
Controller :: index ();

/* 限定名称 */
// 解析为函数 \App\Http\Controllers\Auth\index()
Auth \ index ();
// 解析为类 \App\Http\Controllers\Auth\Controller 的方法 index()
Auth \ Controller :: index ();

/* 完全限定名称 */
// 解析为函数 App\Http\Controllers\index()
\App\Http\Controllers\index ();
// 解析为类 App\Http\Controllers\Auth\Controller 的方法 index()
\App\Http\Controllers\Auth\Controller :: index ();

```

通过上面实例可以看到，在使用类的过程中，可以通过三种方式指定相应的命名空间。有时候由于空间名称过长而导致使用时不够简洁，因此命名空间还允许通过导入外部引用或别名引用的方式指定相应的类和命名空间，即可以为类名称使用别名，也可以为命名空间使用别名。具体实例如下：

```

<?php namespace App\Http\Controllers\Auth;
use App\Http\Controllers\Controller;
use App\Models\Category as DataCategory;
use App\Models;
class AuthController extends Controller
{
    // 其他代码
}
// 实例化 App\Http\Controllers\Controller 对象
$obj1 = new Controller();
// 实例化 App\Http\Controllers\Auth\AuthController 对象
$obj2 = new AuthController();
// 实例化 App\Models\Category 对象
$obj3 = new DataCategory();
// 实例化 App\Models\Records 对象

```

```
$obj4 = new Models\Records();
// 实例化 Models\Records 对象
$obj5 = new \Models\Records();
```

这里需要注意的是，PHP 命名空间只支持导入类，而不支持导入函数或常量，如一个类为 `App\Http\Controllers\Controller`，而该类的命名空间为 `App\Http\Controllers`，就可以通过 `use` 关键字导入该类，也可以导入该命名空间，但该命名空间下的函数或常量是不能通过 `use` 关键字导入的。对命名空间中的名称来说，最前面是不允许有反斜杠的，因为导入的名称都是完全限定的，不会根据命名空间进行相对解析；而在实例化一个对象时，如“`new \Models\Records();`”是可以通过完全限定名称来指定一个类的，此时也不会做相对解析，对于最前面没有反斜杠的实例化类名会进行相对解析。根据前面的实例和叙述，可以得出命名空间名称解析遵循如下规则。

(1) 对完全限定名称的函数、类和常量可以直接解析。例如，`new A\B` 解析为类 `A\B`。

(2) 对所有非限定名称和非完全限定名称的函数、类和常量，根据当前导入的命名空间进行转换。例如，如果命名空间 `A\B\C` 被导入，那么 `new C\D\E()` 就会被转换为 `new A\B\C\D\E()`。

(3) 在命名空间内部，所有的没有根据导入规则转换的非限定名称和非完全限定名称均会在其前面加上当前的命名空间名称。例如，在命名空间 `A\B` 内部调用 `C\D\E()` 时，如果没有导入命名空间 `A\B\C`，则 `new C\D\E()` 会被转换为 `new A\B\C\D\E()`。

(4) 在命名空间内部（例如 `A\B`），对非限定名称和非完全限定名称的函数进行调用时，先在当前命名空间下解析，如果查找不到再在全局空间下查找，即：

1) 在当前命名空间中查找名为 `A\B\foo()` 的函数；

2) 尝试查找并调用全局 (global) 空间中的函数 `foo()`。

(5) 在命名空间（例如 `A\B`）内部对非限定名称和非完全限定名称的类进行调用时，只会在当前命名空间下解析。如对 `new C()`，只会转换为 `new A\B\C()`，如果想引用全局命名空间中的全局类，必须使用完全限定名称 `new \C()`。

3.1.2 文件包含

1. include 和 require 关键字

`Include` 和 `require` 关键字用于包含并运行指定文件。两者作用几乎一样，只是处理失败的方式不同。`require` 在出错时产生 `E_COMPILE_ERROR` 级别的错误，因此会导致脚本程序运行中止，而 `include` 会产生 `E_WARNING` 级别的错误，只会发出警告，而脚本程序会继续运行。

如果包含一个只有文件名的文件，如 `include "file.php"` 时，则系统先会在 PHP 配置文

件中 `include_path` 指定的目录下进行逐一寻找，如果在这些目录下都没有找到，最后会到脚本文件所在的工作目录下寻找，如果依旧未找到，则产生错误，错误级别如上所述。下面是默认情况下 PHP 配置文件中关于 `include_path` 的内容，其中 “.” 表示当前路径。

```
include_path = ".;c:\php\includes"
```

如果包含一个定义了路径的文件，如 `include "../file.php"`，无论是相对路径还是绝对路径，系统只会在相应的路径下寻找该文件，例如一个文件以 “../” 开头，则解析器会在当前目录的父目录下寻找该文件。当一个文件被包含时，包含文件则继承了被包含文件拥有的变量。从该处开始，被包含文件可用的任何变量在包含的文件中也都可用，同时在被包含文件中定义的函数、类或常量都具有全局作用域。

2. 类的自动加载

`include` 和 `require` 关键字是通过手动的方式对相应的文件进行包含，实际上，PHP 提供了更加方便的文件包含方法，即类的自动加载方法。类的自动加载可以通过魔术方法 `__autoload(string $class)` 实现，也可以通过函数 `spl_autoload_register` 注册一个自动加载方法。相应实例如下：

```
function __autoload($class){
    require_once( $class.".php");
}
```

当使用一个类名时，如果该类没有被当前文件包含，则会自动调用 `__autoload($class)` 魔术方法，而其中的 `$class` 为使用类的名称。但在实际应用中，通常使用 `spl_autoload_register` 注册自定义的函数作为自动加载类的实现，因为 `__autoload()` 魔法函数只可以定义一次，而 `spl_autoload_register` 可以将多个类自动加载方法注册到队列中，即创建了 `autoload` 函数的队列，在调用时按照定义时的顺序逐个执行。其函数定义如下：

```
bool spl_autoload_register ([ callable $autoload_function [, bool $throw = true [, bool $prepend = false ]]])
```

通过 `spl_autoload_register()` 函数加载的自动加载函数可以是全局函数，也可以是某个类实例对象的函数，即通过 `array("对象名", "函数名")` 注册。如果没有提供相应的函数参数，则自动注册 `autoload` 的默认实现函数 `spl_autoload()` 为类的自动加载函数。其中，`throw` 参数为 `true` 时，当类的自动加载函数无法成功注册时会抛出异常；当 `prepend` 参数为 `true` 时，`spl_autoload_register()` 会添加类的自动加载函数到队列之首，而不是队列尾部。具体实例如下：

```
public function register($prepend = false)
{
    spl_autoload_register(array($this, 'loadClass'), true, $prepend);
}
```

3. Laravel 中的实现方案

在 Laravel 框架中，通过函数 `spl_autoload_register()` 实现类自动加载函数的注册，其中类的自动加载函数队列中包含了两个类的自动加载函数，一个是 `composer` 生成的基于 PSR 规范的自动加载函数，另一个是 Laravel 框架核心别名的自动加载函数。下面将给出 Laravel 框架中的部分代码，进而介绍该框架下类的自动加载过程。这里 Laravel 框架所在根目录为“`laravel`”，其中 `composer` 生成的自动加载函数注册过程如下：

文件：`laravel\public\index.php`

```
<?php
require __DIR__.'../../bootstrap/autoload.php';
```

文件：`laravel\bootstrap\autoload.php`

```
<?php
define('LARAVEL_START', microtime(true));
require __DIR__.'../../vendor/autoload.php';
```

在 Laravel 框架中，`public\index.php` 文件为请求的入口文件，其中第一句便是包含启动文件夹下的自动加载的文件，而该文件继续包含 `vendor` 目录下的自动加载文件，其中 `vendor` 目录是 `composer` 生成的依赖包目录，而内部的自动加载文件也是 `composer` 生成的，用于自动加载依赖包中的所有文件。下面简单介绍 `composer` 生成的类自动加载函数是如何实现的。

文件：`laravel\vendor\autoload.php`

```
<?php
require_once __DIR__ . '/composer' . '/autoload_real.php';
return ComposerAutoloaderInit99123d508294c719fdcf537b9ee84731::getLoad
er();
```

文件：`laravel\vendor\composer\autoload_real.php`

```
<?php
class ComposerAutoloaderInit99123d508294c719fdcf537b9ee84731
{
    private static $loader;
    public static function loadClassLoader($class)
    {
        if ('Composer\Autoload\ClassLoader' === $class) {
            require __DIR__ . '/ClassLoader.php';
        }
    }
    public static function getLoader()
```

```

    {
        if (null !== self::$loader) {
            return self::$loader;
        }

        spl_autoload_register(array('ComposerAutoloaderInit99123d508294c719fdcf537b9ee84731', 'loadClassLoader'), true, true);

        self::$loader = $loader = new \Composer\Autoload\ClassLoader();

        spl_autoload_unregister(array('ComposerAutoloaderInit99123d508294c719fdcf537b9ee84731', 'loadClassLoader'));

        $map = require __DIR__ . '/autoload_namespaces.php';
        foreach ($map as $namespace => $path) {
            $loader->set($namespace, $path);
        }

        $map = require __DIR__ . '/autoload_psr4.php';
        foreach ($map as $namespace => $path) {
            $loader->setPsr4($namespace, $path);
        }

        $classMap = require __DIR__ . '/autoload_classmap.php';
        if ($classMap) {
            $loader->addClassMap($classMap);
        }

        $loader->register(true); // 注册类自动加载函数

        $includeFiles = require __DIR__ . '/autoload_files.php';
        foreach ($includeFiles as $file) {
            composerRequire99123d508294c719fdcf537b9ee84731($file);
        }

        return $loader;
    }
}

function composerRequire99123d508294c719fdcf537b9ee84731($file)
{
    require $file;
}

```

通过 composer 工具创建依赖管理时，会在 vendor 目录下创建一个 autoload.php 文件和一个 composer 文件夹，其中 composer 文件夹中包含了类自动加载函数注册的相关实现，而 autoload.php 文件是对外提供的接口，通过包含该文件就可以完成类自动加载函数的注册。通过上述代码可以看到，autoload.php 文件包含了 composer 目录下的 autoload_real.php 文件，而 autoload_real.php 文件定义了一个类，该类由末尾有一串数字的方式定义，并且定义了 getLoader() 函数，该函数首先实例化 Composer\Autoload\ClassLoader 类，然后通过该类实例添加相关的文件路径配置，包括命名空间 (autoload_namespaces.php 文件定义) 配置、PSR-4 规范 (autoload_psr4.php 文件定义) 配置、类映射 (autoload_classmap.php 文件定义) 配置，

接着调用 \$loader → register(true) 注册类自动加载函数，最后加载全局文件，即在 autoload_files.php 文件中配置的内容。下面介绍如何注册类自动加载函数，以及类自动加载函数是如何实现类的自动加载的。

文件：laravel\vendor\composer\ClassLoader.php

```
<?php
namespace Composer\Autoload;
class ClassLoader
{
    // 省略加载文件路径函数的相关代码
    public function register($prepend = false)
    {
        spl_autoload_register(array($this, 'loadClass'), true, $prepend);
    }
    public function loadClass($class)
    {
        if ($file = $this->findFile($class)) {
            includeFile($file);
            return true;
        }
    }
    // 省略了根据加载文件路径查找文件具体位置的相关代码
}
```

通过前面的介绍，已经了解了类的自动加载函数是在 Composer\Autoload\ClassLoader 类中实现的，实例化该类并将类的命名空间与文件路径的对应关系注册到相应属性中，然后通过实例方法 register(\$prepend = false) 注册一个类自动加载函数，即为该类实例的 loadClass 方法，并且将其注册在类自动加载函数队列的末尾，当使用一个为包含的类名时，会自动调用 loadClass 方法并通过参数获取包含命名空间的类名信息，接着根据类的命名空间与文件路径的对应关系查找文件路径，最后通过 includeFile() 函数包含该文件，实现类的自动加载。

前文提到过，默认 Laravel 框架包含两个类的自动加载函数，其中一个是在外观注册 (Illuminate\Foundation\Bootstrap\RegisterFacades 类实现的) 过程中实现的，这里只需了解有这样一个类自动加载函数被注册到堆栈中就可以，后面可以在“请求到响应的生命周期”内容中了解调用过程，这里只介绍类的自动加载函数的注册过程。在注册过程中也是先实例化后调用 register() 函数，进而调用 prependToLoaderStack() 函数，将 load(\$alias) 函数注册为类的自动加载函数，该函数的作用主要是将外观别名与外观名 (Facades) 对应起来，从而实现对应外观类的静态方法调用。对应类的自动加载函数注册过程实现代码如下：

文件: Illuminate\Foundation\AliasLoader.php

```
<?php
namespace Illuminate\Foundation;
class AliasLoader
{
    // 加载一个类别名, 实际上是给外观类起了一个别名, 使两者对应一个类
    public function load($alias)
    {
        if (isset($this->aliases[$alias])) {
            return class_alias($this->aliases[$alias], $alias);
        }
    }
    // 添加别名到自动加载函数中
    public function alias($class, $alias)
    {
        $this->aliases[$class] = $alias;
    }
    // 注册自动加载函数到自动加载堆栈中
    public function register()
    {
        if (!$this->registered) {
            $this->prependToLoaderStack();
            $this->registered = true;
        }
    }
    // 将类的自动加载函数添加到自动加载堆栈首部
    protected function prependToLoaderStack()
    {
        spl_autoload_register([$this, 'load'], true, true);
    }
}
```

3.2 匿名函数

匿名函数(Anonymous functions)也叫闭包函数(Closure), 即一个没有指定名称的函数, 经常用做回调函数(callback)参数的值。当然, 也有其他应用的情况。Closure(闭包)类也称匿名函数类, 匿名函数(在 PHP 5.3 中被引入)本身就是这个类型的对象。刚开始, 大家只是将匿名函数当做该类的一个实现, 但自 PHP 5.4 之后, 闭包类逐渐添加了一些方法, 允许在匿名函数创建后对其进行更多的控制, 使得匿名函数类的应用更加灵活, 在 Laravel 框架中, 大量地使用了匿名函数, 使得框架更加紧凑、灵活。

3.2.1 匿名函数的使用

通常我们在调用函数时，传入的参数是数据，那么只能通过参数对函数的结果进行控制，无法控制其过程，而匿名函数的存在既可以作为参数传给函数，也可以作为变量赋值，进而控制函数的执行过程，因此，匿名函数的引入增强了程序编写的灵活性，可以实现更加高效的设计方案。下面给出相应的实例：

```
<?php
$array = array(1, 2, 3, 4);
//array_walk 使用用户自定义函数对数组中的每个元素做回调处理
array_walk($array, function($value){echo $value;});
输出： 1 2 3 4
```

匿名函数的另一个作用是可以从父作用域中继承变量，即匿名函数在定义的时候如果需要使用作用域外的变量，可以使用 `use` 关键字来继承作用域外的变量，具体实例如下：

```
<?php
function getCounter() {
    $i = 0;
    return function() use($i) {
        echo ++$i;
    };
}
$counter = getCounter();
$counter();
$counter();
输出： 1 1
```

匿名函数在每次执行的时候都能访问到上层作用域内的变量，这些变量在匿名函数被销毁之前始终保存着自己的状态。这里两次函数调用并没有使 `$i` 变量自增，因为默认 PHP 是通过复制的方式传入上层变量进入匿名函数，如果需要改变上层变量的值，则需要通过引用的方式传递，即 `use(&$i)`。所以上面的代码没有输出 1 和 2，而是输出 1 和 1。

3.2.2 Laravel 框架中的应用

在 Laravel 框架中大量地使用了匿名函数，如在服务提供者注册过程中，通过将服务名称与提供服务的匿名函数进行绑定，在使用时可以实现动态服务解析。这里所有的服务可以通俗地理解为对一种资源的提供，这个资源可以是一个类的实例、一个路径或是一个文件等，提供服务就是提供一种资源。具体实例如下：

文件 `Illuminate\Routing\ControllerServiceProvider.php`

```
<?php
```



```
namespace Illuminate\Routing;
use Illuminate\Support\ServiceProvider;
class ControllerServiceProvider extends ServiceProvider
{
    // 注册服务提供者
    public function register()
    {
        $this->app->singleton('illuminate.route.dispatcher', function
($app) {
            return new ControllerDispatcher($app['router'], $app);
        });
    }
}
```

从上面例子可以看出，这里的 `$this → app → singleton()` 函数的作用是将服务名 `illuminate.route.dispatcher` 与后面的提供服务的匿名函数绑定起来，用于服务解析，服务就是通过匿名函数实现的。

3.3 PHP 中的特殊语法

PHP 中有一些特殊的方法和常量，即魔术方法和魔术常量。魔术方法和魔术常量的主要目的是提供对 PHP 运行环境和过程的控制和检测。

3.3.1 魔术方法

魔术方法与普通方法相比具有很大的不同，普通方法是根据用户的实现方式进行调用，而魔术方法通常情况下用户不会主动调用，而是在特定的时机被 PHP 系统自动调用，可以通俗地理解为系统事件监听方法，在事件发生时才触发执行，与嵌入式系统中的中断函数类似。在 PHP 中通常以 “`__`” 打头的方法都作为魔术方法，所以用户不要定义以 “`__`” 开头的方法。例如，类的构造方法 `__construct()`，该方法是在创建实例完成后自动调用的一个方法。

对于魔术方法的使用需要了解两个方面内容，一是魔术函数定义的位置，二是魔术函数调用的时机。其中，PHP 中常用的魔术方法的使用方法如表 3.1 所示。

表 3.1 常用的魔术方法的使用方法

魔 术 方 法	使 用 方 法
<code>__construct()</code>	构造函数会在每次实例化对象时先调用此方法，所以非常适合在使用对象之前做一些初始化工作
<code>__destruct()</code>	析构函数会在某个对象的所有引用都被删除或者当对象被显式销毁时执行
<code>__set()</code>	用于属性重载，在给不可访问的属性赋值时， <code>__set()</code> 会被调用

续表

魔 术 方 法	使 用 方 法
__get()	读取不可访问属性的值时，__get() 会被调用
__isset()	当对不可访问属性调用 isset() 或 empty() 时，__isset() 会被调用
__unset()	当对不可访问属性调用 unset() 时，__unset() 会被调用
__sleep()	serialize() 检查类中是否有魔术方法 __sleep()。如果存在，该函数将在任何序列化之前运行。它可以清除对象并返回一个包含有该对象中应被序列化的所有变量名的数组
__wakeup()	相反地，unserialize() 检查是否具有魔术方法 __wakeup()。如果存在，此函数可以用于重建对象
__toString()	用于一个类被当成字符串时应怎样回应。例如，执行“echo \$obj;”语句时，该方法会被调用
__invoke()	当尝试以调用函数的方式调用一个对象时，__invoke() 方法会被自动调用。例如，执行“\$obj();”语句时，该方法会被调用
__clone()	如果定义了 __clone() 魔术方法，则新创建对象（复制生成对象）时，__clone() 方法会被调用，可用于修改属性的值
__call()	在对象中调用一个不可访问的方法时，__call() 会被调用
__callStatic()	在静态方式中调用一个不可访问的方法时，__callStatic() 会被调用
__autoload()	它会在试图使用尚未被定义的类时自动调用

下面重点介绍在 Laravel 框架中常用的几个魔术方法。通过 Laravel 框架中 Illuminate\Database\Eloquent\Model 类中的代码作为实例进行介绍。

文件：Illuminate\Database\Eloquent\Model.php

```
<?php
namespace Illuminate\Database\Eloquent;

// 省略命名空间引用部分代码

abstract class Model implements ArrayAccess, Arrayable, Jsonable,
JsonSerializable, QueueableEntity, UrlRoutable
{
    // 创建一个新的 Eloquent 模型实例
    public function __construct(array $attributes = [])
    {
        $this->bootIfNotBooted();
        $this->syncOriginal();
        $this->fill($attributes);
    }
    // 在模型实例中索引不存在的属性
    public function __get($key)
    {
        return $this->getAttribute($key);
    }
}
```

```

// 在模型实例中设置不存在的属性
public function __set($key, $value)
{
    $this->setAttribute($key, $value);
}

// 判断在模型实例中是否存在该属性
public function __isset($key)
{
    return (isset($this->attributes[$key]) || isset($this->relations[$key])) ||
        ($this->hasGetMutator($key) && !is_null($this->getAttributeValue($key)));
}

// 注销模型实例中的一个属性
public function __unset($key)
{
    unset($this->attributes[$key], $this->relations[$key]);
}

// 处理模型实例中不存在的函数调用
public function __call($method, $parameters)
{
    if (in_array($method, ['increment', 'decrement'])) {
        return call_user_func_array([$this, $method], $parameters);
    }
    $query = $this->newQuery();
    return call_user_func_array([$query, $method], $parameters);
}

// 处理模型实例中不存在的静态函数调用
public static function __callStatic($method, $parameters)
{
    $instance = new static;
    return call_user_func_array([$instance, $method], $parameters);
}

// 将模型实例转换为替代的字符串
public function __toString()
{
    return $this->toJson();
}

// 当模型类被反序列化时, 判断是否需要启动
public function __wakeup()
{
    $this->bootIfNotBooted();
}
}

```


为了测试上述各魔术方法的调用时机，这里将 Laravel 框架的入口文件 `laravel/public/index.php` 进行修改，修改的程序代码如下所示。其中前面三行为 Laravel 框架环境启动部分，这部分内容将在请求的生命周期中详细介绍。接着通过命名空间引入 Model 类，创建了 User 类，该类继承了 Model 类，然后获取对象 \$user，并触发相应的魔术方法调用。注意，这里对 `find()` 方法的调用需要数据库的支持。

文件：laravel/public/index.php

```
<?php
require __DIR__.'../bootstrap/autoload.php';
$app = require_once __DIR__.'../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
    use Illuminate\Database\Eloquent\Model;
    // 引入 Model 类
class User extends Model
{
    // 定义 User 类并继承 Model 类，声明数据库表为 'users'。
    protected $table = 'users';
}
    // __construct 将被调用，用于初始化对象
$user = new User();
    // __set($key,$value) 将被调用，$key="name"，$value="xiaozhang"，用于属性设置
$user->name = 'xiaozhang';
    // __get($key) 将被调用，$key="name"，用于属性获取
echo $user->name;
    // __isset($key) 将被调用，$key="name"，用于判断属性是否存在
isset($user->name);
    // __unset($key) 将被调用，$key="name"
unset($user->name);
    // __call($method,$parameters) 将被调用，$method="find" $parameters = 5
$user->find(1);
    // __callStatic($method,$parameters) 将被调用，$method="find" $parameters
= 5
User::find(1);
    // __toString() 将被调用
echo $user;
$user = serialize($user);
    // __wakeup() 将被调用
$test2 = unserialize($us);
```

3.3.2 魔术常量

PHP 向它运行的任何脚本提供了很多预定义常量。不过很多常量都是由不同的扩展库

定义的，只有在加载了这些扩展库时才会出现，或者动态加载后，或者在编译时已经包括进去了。同时，PHP 也在运行环境中提供了八个魔术常量，它们虽然被称为魔术常量，但是它们的值随着代码中的位置改变而改变。例如，`__LINE__` 的值就依赖于它在脚本中所处的行来决定。PHP 中常用的魔术常量使用方法如表 3.2 所示。

表 3.2 常用的魔术常量使用方法

名 称	使 用 方 法
<code>__LINE__</code>	文件中的当前行号
<code>__FILE__</code>	文件的完整路径和文件名。如果用在被包含文件中，则返回被包含的文件名。自 PHP 4.0.2 起， <code>__FILE__</code> 总是包含一个绝对路径，而在此之前的版本有时会包含一个相对路径
<code>__DIR__</code>	文件所在的目录。如果用在被包含文件中，则返回被包含的文件所在的目录。它等价于 <code>dirname(__FILE__)</code> 。除非是根目录，否则目录中名不包括末尾的斜杠
<code>__FUNCTION__</code>	函数名称（PHP 4.3.0 新加的魔术常量）。自 PHP 5 起本常量返回该函数被定义时的名字并区分大小写。在 PHP 4 中该值总是小写字母
<code>__CLASS__</code>	类的名称（PHP 4.3.0 新加的魔术常量）。自 PHP 5 起本常量返回该类被定义时的名字并区分大小写。在 PHP 4 中该值总是小写字母。类名包括其被声明的命名空间（如 <code>Foo\Bar</code> ）。注意自 PHP 5.4 起， <code>__CLASS__</code> 对 <code>trait</code> 也起作用。当用在 <code>trait</code> 方法中时， <code>__CLASS__</code> 是指调用 <code>trait</code> 方法的类的名字
<code>__TRAIT__</code>	<code>trait</code> 的名字（PHP 5.4.0 新加的魔术常量）。自 PHP 5.4 起，此常量返回 <code>trait</code> 被定义时的名字并区分大小写， <code>trait</code> 名包括其被声明的命名空间（如 <code>Foo\TraitBar</code> ）
<code>__METHOD__</code>	类的方法名（PHP 5.0.0 新加的魔术常量）。返回该方法被定义时的名字并区分大小写
<code>__NAMESPACE__</code>	返回当前命名空间的名称并区分大小写（PHP 5.3.0 新加的魔术常量）

在 Laravel 框架中，应用最多的是 `__DIR__` 常量，用于确定文件所在的目录，通过严格对应的目录结构，可以找到其他文件所在的位置。这里使用 Laravel 框架的入口文件作为实例，如下所示：

文件：`laravel\public\index.php`

```
<?php
require __DIR__.'../bootstrap/autoload.php';
$app = require_once __DIR__.'../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);
$response->send();
$kernel->terminate($request, $response);
```

该文件为 Laravel 框架的入口文件，即 `laravel\public\index.php` 文件，该文件中根据 `__`

`DIR__` 获得本文件的目录，再根据目录的相对位置来启动其他文件代码。

3.4 反射

反射机制被 Ruby、PHP 等多种语言广泛应用，主要用来动态地获取系统中类、实例对象、方法等语言构件的信息，通过反射 API 函数可以实现对这些语言构件信息的动态获取和动态操作等。PHP 5 具有完整的反射 API，添加了对类、接口、函数、方法和扩展进行反向工作的能力。此外，反射 API 还提供了获取函数、类和方法等语言构件中的文档的注释方法。下面介绍一个具体实例。

```
<?php
class A {
    public function call()
    {
        echo "Hello wshuo";
    }
}

$ref = new ReflectionClass('A');
$inst = $ref->newInstanceArgs();
$inst->call();
输出: Hello wshuo
```

通过上面的实例可以看到反射机制的强大，在很多情况下可以使得代码更加高效，例如事先不知道需要实例化哪个类，而是在运行时根据动态信息确定，对于这种情况可以通过反射机制获取需要实例化类的构造函数信息并完成相应的实例化。在 Laravel 框架中，服务容器解析服务的过程中就用到了反射机制。下面给出 Laravel 框架中解析服务的实例。

文件: `Illuminate\Container\Container.php`

```
// 根据给定的类初始化一个具体实例
public function build($concrete, array $parameters = [])
{
    if ($concrete instanceof Closure) {
        return $concrete($this, $parameters);
    }

    $reflector = new ReflectionClass($concrete);
    if (! $reflector->isInstantiable()) {
        $message = "Target [$concrete] is not instantiable.";
        throw new BindingResolutionContractException($message);
    }

    $this->buildStack[] = $concrete;
```



```

    $constructor = $reflector->getConstructor();
    if (is_null($constructor)) {
        array_pop($this->buildStack);
        return new $concrete;
    }
    $dependencies = $constructor->getParameters();
    $parameters = $this->keyParametersByArgument(
        $dependencies, $parameters
    );
    $instances = $this->getDependencies(
        $dependencies, $parameters
    );
    array_pop($this->buildStack);
    return $reflector->newInstanceArgs($instances);
}

```

在 Laravel 框架中，解析服务是通过 build() 函数实现的，一般分为两种情况：一种是查找对应服务是否被服务提供者注册为实例或提供服务的匿名函数，如果是，则直接进行服务解析；第二种是服务名称没有相应的服务绑定，通过反射机制来动态创建服务。通过反射机制动态创建服务的过程可以分为两个步骤：第一步是通过反射机制获取服务类构造函数的信息，第二步是解决服务类构造函数的依赖问题。首先，通过 “\$reflector = new ReflectionClass(\$concrete);” 来创建一个反射类实例，其中 \$concrete 是类的名称，然后通过 “\$reflector->isInstantiable();” 判断这个类是否可以实例化，如果不可以则抛出异常，接着通过 “\$constructor = \$reflector->getConstructor();” 来获取类的构造方法，当该类存在构造函数时，返回一个 ReflectionMethod 对象，相当于获取构造函数的反射类，当类不存在构造函数时返回 NULL，最后通过 “is_null(\$constructor)” 判断是否存在构造函数，如果不存在则直接实例化该类，如果存在则通过 “\$dependencies = \$constructor->getParameters();” 来获取构造函数依赖的输入参数。下面将解决构造函数中依赖参数的问题，进而实现依赖注入。

文件：Illuminate\Container\Container.php

```

// 根据反射参数解决所有的参数依赖
protected function getDependencies(array $parameters, array $primitives = [])
{
    $dependencies = [];
    foreach ($parameters as $parameter) {
        $dependency = $parameter->getClass();
        if (array_key_exists($parameter->name, $primitives)) {
            $dependencies[] = $primitives[$parameter->name];
        } elseif (is_null($dependency)) {
            $dependencies[] = $this->resolveNonClass($parameter);
        }
    }
}

```

```

    } else {
        $dependencies[] = $this->resolveClass($parameter);
    }
}

return (array) $dependencies;
}

// 解决无法获取类名的依赖
protected function resolveNonClass(ReflectionParameter $parameter)
{
    if ($parameter->isDefaultValueAvailable()) {
        return $parameter->getDefaultValue();
    }

    // 省略异常处理部分代码
}

// 通过服务容器解决一个具有类名的依赖
protected function resolveClass(ReflectionParameter $parameter)
{
    try {
        return $this->make($parameter->getClass()->name);
    }

    // 省略异常处理部分代码
}

```

依赖和依赖注入的概念将会在 Laravel 框架中的设计模式章节详细介绍，这里可以简单地理解为获取类构造函数中的参数，进而完成类的实例化。首先，通过“`$parameters = $this->keyParametersByArgument($dependencies, $parameters);`”获取直接提供的实参，未直接提供的通过“`$instances = $this->getDependencies($dependencies, $parameters);`”根据形参的类型取实参。在 `getDependencies()` 函数中需要调用 `resolveNonClass()` 函数或 `resolveClass()` 函数解决参数依赖问题，对于构造函数的参数，如果无法获取该参数的类名，则通过 `resolveNonClass()` 函数获取默认的参数值，如果可以获取类的名称，则通过 `resolveClass()` 函数进行实例化，而实例化过程是通过服务容器进行解析的，即通过“`$parameter->getClass()->name`”来获取参数的类名，然后通过“`$this->make($parameter->getClass()->name);`”来解析服务，`make()` 函数接下来还会调用 `build()` 函数完成类的实例化过程，相当于一个递归调用的过程，最终由“`$reflector->newInstanceArgs($instances);`”实例化服务类，进而完成服务的解析。这部分内容涉及到很多新的概念，如依赖注入、服务容器、服务解析等，这些概念将会在后面的章节中进行详细介绍，这里读者只需要对反射机制有一个了解就可以了，随着介绍的深入会真正理解其本质。

3.5 后期静态绑定

从 PHP 5.3.0 开始，PHP 增加了一个叫做后期静态绑定的功能，用于在继承范围内引用静态调用的类，即在类的继承过程中，使用的类不再是当前类，而是调用的类。后期静态绑定使用关键字 `static` 来实现，通过这种机制，“`static::`”不再被解析为定义当前方法所在的类，而是在实际运行时计算得到的，即为运行时最初调用的类。虽然将其称之为“后期静态绑定”，但它不仅限于静态方法的调用。下面介绍一个具体实例。

```
<?php
class A {
    public static function call () {
        echo "class A."<br>" ;
    }
    public static function test () {
        self :: call ();
        static::call();
    }
}
class B extends A {
    public static function call () {
        echo "class B."<br>" ;
    }
}
B :: test ();
输出: class A
      class B
```

通过上述实例可以看出，在调用 `test()` 函数时，“`self::`”是直接调用本类中的方法，而 `static` 是根据调用 `test()` 函数的类来决定“`static::`”的值，因此 `static` 的值只有在调用时才能确定下来，而 `self` 则是在定义时就确定下来的。需要注意的是，`static` 并不限于静态方法调用，同样适用于非静态函数的调用，调用的方式同上述静态函数一样，是在调用时动态确定的。下面给出非静态方法调用的实例。

```
class A {
    public function call () {
        echo "instance from A."<br>" ;
    }
    public function test () {
        self::call ();
        static:: call ();
    }
}
```



```

class B extends A {
    public function call(){
        echo "instance from B"."<br>";
    }
}
$b = new B ();
$b->test ();
输出: instance from A
      instance from B

```

后期静态绑定还可以用于对象实例化中，同上述内容一样，在实例化对象时，static 会根据运行时调用的类来决定实例化对象，而 self 是根据所在位置的类来决定实例化对象，下面给出一个具体实例。

```

<?php
class A
{
    public static function create()
    {
        $self = new self();
        $static = new static();
        return array($self,$static);
    }
}
class B extends A
{
}
$arr = B::create();
foreach($arr as $value){
    var_dump($value);
}
输出: object(A) [1]
      object(B) [2]

```

在 Laravel 框架中，以上三种使用方法经常可以遇到，下面是 Laravel 框架中 Illuminate\Database\Eloquent\Model 类的部分代码，该类中大量使用了后期静态绑定，如 “\$model = new static(\$attributes);” 和 “return static::create(\$attributes);” 等，因为该类为抽象类，所以它的实现类在调用这些函数时，最终动态绑定的都是实现类，而非这个 Model 抽象类。

文件: Illuminate\Database\Eloquent\Model.php

```

// 保存模型类实例中的数据并返回该模型类实例
public static function create(array $attributes = [])
{
    $model = new static($attributes);
}

```

```

        $model->save();
        return $model;
    }
    // 获取第一个符合条件的数据或创建一个新的
    public static function firstOrCreate(array $attributes)
    {
        if (! is_null($instance = static::where($attributes)->first())) {
            return $instance;
        }
        return static::create($attributes);
    }
}

```

3.6 Laravel 中使用的其他新特性

3.6.1 trait

相对于多继承语言（如 C++），代码复用这个问题对于单继承类语言（如 Ruby、PHP 等）来说需要通过其他方法来解决，例如 Ruby 中通过混入类（Mixin）的方法来解决。PHP 自 5.4.0 起，使用了一种简洁的方案来实现代码复用，即 trait。

一个 trait 和一个类相似，但 trait 不能像类一样进行实例化，而是通过关键字 use 添加到其他类的内部，从而发挥它的作用。相对于传统继承方法，trait 增加了水平特性的组合。下面介绍一个具体实例。

```

<?php
class Base
{
    public function hello ()
    {
        echo 'method hello from class Base'.<br>';
    }
}

trait Hello
{
    public function hello ()
    {
        echo 'method hello from Trait Hello!'.<br>';
    }
    public function hi()
    {
        echo 'method hi from Trait Hello'.<br>';
    }
}

```

```

abstract public function getValue();
static public function staticMethod()
{
    echo 'static method staticMethod from Trait Hello'.<br>';
}
public function staticValue()
{
    static $value;
    $value++;
    echo "$value".<br>';
}
}

trait Hi
{
    public function hello ()
    {
        parent :: hello ();
        echo 'method hello from Trait Hi!'.<br>';
    }
    public function hi()
    {
        echo 'method hi from Trait Hi!'.<br>';
    }
}

trait HelloHi
{
    use Hello, Hi{
        Hello::hello insteadof Hi;
        Hi::hi insteadof Hello;
    }
}

class MyNew extends Base
{
    use HelloHi;
    private $value = 'class MyNew'.<br>';
    public function hi()
    {
        echo 'method hi from class MyNew'.<br>';
    }
    public function getValue()
    {

```



```

        return $this->value;
    }
}

$obj = new MyNew ();
$obj->hello ();
// 输出: 'method hello from Trait Hello!'
// 优先级顺序: trait 中的方法又覆盖了基类中的方法
$obj->hi();
// 输出: 'method hi from class MyNew'
// 优先级顺序: 当前类中的方法会覆盖 trait 方法
MyNew::staticMethod();
// 输出: 'static method staticMethod from Trait Hello'
// 静态方法: trait 中可以定义静态方法
echo $obj->getValue();
// 输出: 'class MyNew'
// 抽象成员: trait 中可以使用抽象方法
$objOther = new MyNew();
$obj->staticValue();
// 输出: 1
// 静态成员: trait 中可以使用静态成员
$objOther->staticValue();
// 输出: 2

```

为了在一个实例中介绍更多关于 trait 的性质，上面的实例相对来说有些复杂。根据上面的实例，下面列出一些 trait 的重要性质。

- (1) 优先级：当前类的方法会覆盖 trait 中的方法，而 trait 中的方法会覆盖基类的方法。
- (2) 多个 trait 组合：通过逗号分隔，通过 use 关键字列出多个 trait。
- (3) 冲突的解决：如果两个 trait 都插入了一个同名的方法，若没有明确解决冲突将会产生一个致命错误。为了解决多个 trait 在同一个类中的命名冲突，需要使用 insteadof 操作符来明确指定使用冲突方法中的哪一个。同时，可以通过 as 操作符将其中一个冲突的方法以另一个名称来引入。

- (4) 修改方法的访问控制：使用 as 语法可以用来调整方法的访问控制。
- (5) trait 的抽象方法：在 trait 中可以使用抽象成员，使得类中必须实现这个抽象方法。
- (6) trait 的静态成员：在 trait 中可以用静态方法和静态变量。
- (7) trait 的属性定义：在 trait 中同样可以定义属性。

在 Laravel 框架中很多地方也应用到了 trait，下面是身份认证中的部分代码。

文件：laravel\app\Http\Controllers\Auth\AuthController.php

```

class AuthController extends Controller
{

```

```
use AuthenticatesAndRegistersUsers, ThrottlesLogins;
// 省略其他代码
}
```

文件：Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers.php

```
trait AuthenticatesAndRegistersUsers
{
    use AuthenticatesUsers, RegistersUsers {
        AuthenticatesUsers::redirectPath insteadof RegistersUsers;
    }
}
```

文件：Illuminate\Foundation\Auth\AuthenticatesUsers.php

```
trait AuthenticatesUsers
{
    use RedirectsUsers;
    // 省略具体代码部分
}
```

文件：Illuminate\Foundation\Auth\RegistersUsers.php

```
trait RegistersUsers
{
    use RedirectsUsers;
    // 省略具体代码部分
}
```

在文件“AuthenticatesUsers.php”和“RegistersUsers.php”中通过 trait 形式分别定义了登录认证和注册的相关函数，在文件“AuthenticatesAndRegistersUsers.php”中通过 trait 组合方式得到登录认证和注册的组合，最后在身份控制文件“AuthController.php”中使用了这个 trait，如果系统有多个登录注册模块，可以重复使用这些 trait，进而达到代码复用的目的。

3.6.2 简化的三元运算符

三元运算符“?:”的通常用法是“\$value = (expr1) ? (expr2) : (expr3)”。当表达式“expr1”求值为 true 时，“\$value”值为“expr2”；当表达式“expr1”求值为 false 时，“\$value”值为“expr3”。

自 PHP 5.3 起，可以省略三元运算符的中间部分，得到三元运算符的简化形式“\$value = expr1 ?: expr3”。当“expr1”求值为 true 时，“\$value”值为 expr1，否则“\$value”值为“expr3”。

本章介绍了 PHP 语言的一些新的语法特性，每一种新语法特性的产生都预示着一种新的开发模式的诞生，而这种开发模式也将带来它的优秀特性，比如命名空间为组件化开发提供支撑、反射机制为服务容器的实现提供支撑等，如果没有这些新的特性，Laravel 框架可能无法达到现在的优美程度，所以掌握这些新的语法特性是理解 Laravel 框架必须做的课前工作。

第 4 章

Laravel 框架中使用的 HTTP 协议基础

在 Web 开发中，HTTP 协议作为客户端和服务端交互的准则实现客户请求的发送及响应信息的获取，在构建 B/S 结构应用中，前端工程师专注于酷炫多彩的页面实现、功能强大的 UI 框架设计等，后端工程师专注于复杂的业务流程、可扩展—低耦合的架构设计等，因此常常忽略了两端之间交互的环节，即 HTTP 协议部分。然而，HTTP 协议作为 Web 开发的基础，无论是前端还是后端，工程师只有掌握了这部分内容才能在软件开发这条道路上走得更远、更坚实，才能抓住一些问题的本质进行解决。同时，也只有了解 HTTP 协议的基本内容，才能更好地掌握 Laravel 框架，因为该框架中无论是请求获取、响应生成还是 session 实现等内容都和 HTTP 协议密切相关。本章将概要地介绍 HTTP 协议的内容，如需更加深入地了解，读者可以参看 HTTP 相关的书籍。

4.1 HTTP 发展与相关网络技术

通常情况下，在浏览器地址栏中输入一个正确的网络地址就会获得一个网页显示，这种功能是如何实现的呢？首先，可以明确的是网页并不是本机上的，而是存储在互联网的某一台服务器上，那么就需要解决三个关键问题：第一是获取内容所在目标服务器，第二是相互间传递消息，第三就是解析服务器返回的内容。获取目标服务器地址通过统一资源定位符（Uniform Resource Locator，即 URL）实现，相互间信息的传输通过超文本传输协议（HyperText Transfer Protocol，即 HTTP）实现，而服务器返回内容的解析通过超文本标记语言（HyperText Markup Language，即 HTML）实现。下面将围绕这三个问题介绍 HTTP 的发展和相关网络技术。

4.1.1 HTTP 发展

HTTP 协议的最初设想是 Tim Berners Lee 于 1989 年提出的，当时用来解决不同地域的研究学者实现技术知识的共享问题，而在 1990 年即诞生了世界上的第一台服务器和浏览器。在随后的二十几年中，HTTP 协议获得了长足的发展，主要经历了三个版本：第一个版本是

HTTP 协议诞生到正式标准的建立, 这个阶段的版本被统称为 HTTP/0.9 版本; 第二个版本是 1996 年 5 月正式公布的标准版本, 被命名为 HTTP/1.0 版本, 标准规范由 RFC1945 记录; 第三个版本是 1997 年 1 月公布的 HTTP/1.1 版本, 标准规范由 RFC2068 记录。目前大部分服务器都使用 HTTP/1.0 和 HTTP/1.1 版本, 对于目前正在发展的 HTTP/2.0 版本还需要一定的时间进行普及。

不同版本的 HTTP 协议发展是与各阶段 Web 需求分不开的, 起初 HTTP 协议主要为了实现文本资料的共享, 所以 HTTP/0.9 的请求只有一行, 只支持 GET 方法, 只能接收 HTML 文本的响应。后来, 为了实现客户端和服务端进行数据交互, HTTP/1.0 加入了 POST 等方法, 同时 HTTP 协议传输的内容也扩展到图片、动画等格式。接着, 为了提升在 HTTP 协议下工作的服务器的性能, HTTP/1.1 在 HTTP/1.0 的基础上进行了增强, 增加了包括默认长连接保持等功能。

4.1.2 与 HTTP 协议相关的网络技术

HTTP 协议是一种应用层协议, 它的实现离不开其他的网络技术, 与其关系密切的就有 TCP、IP、ARP、DNS 等。

对于网络技术, 最常听到的就是 TCP/IP 协议, 实际上 TCP/IP 协议并不是指某种协议, 而是为实现计算机间通信的网络体系。在计算机通信的发展过程中, 诞生了很多网络体系, 包括 TCP/IP、OSI、IPX/SPX 等, 每一种网络体系中包含很多相应的协议, 这些协议的集合被称为协议族, 其中 TCP/IP 协议族标准是由国际互联网工程任务组 (The Internet Engineering Task Force, 简称 IETF) 制定的, 包括 IP、TCP、UDP、HTTP、ARP 等协议。OSI (Open Systems Interconnection, 即开放式通信系统互联) 协议族标准是由国际标准化组织 (International Organization for Standards, 简称 ISO) 制定的, 包括 FTAM、VT、CONP 等协议。IPX/SPX 协议族标准是由 Novell 公司制定的, 包括 IPX、SPX、NPC 等协议。目前, 经常接触到的是 TCP/IP 网络体系和 OSI 网络体系, 其中 OSI 所定义的协议虽然没有得到普遍应用, 但是其设计的 OSI 参考模型经常被用于网络协议制定, 而 TCP/IP 已经成为全世界范围内共同遵循应用的一种通信协议标准, 该标准也使得不同硬件和系统的网络设备可以进行互相通信。

为了使复杂的网络协议变得简单化, 协议通常被分层处理, OSI 协议被分为七层, 而 TCP/IP 协议被分为五层。在这种分层模型中, 每一层的实现都需要下一层提供特定的服务, 而本层实现的功能又为上一层提供服务, 层与层之间遵循“接口”的约定, 而两台机器同层间遵循“协议”的约定。每一层会将上一层传过来的数据附加一个首部, 而加上首部的数据被送到下一层, 下一层会将接收到的数据包当成整个数据并加入本层的首部, 每一层首部至少都包含两部分信息, 一部分是接收者和发送者的地址, 另一部分是上一层的协议类型。对于接收者和发送者的地址, 数据链路层使用 MAC 地址、网络层使用 IP 地址、传输层使用端口号。结合协议分层模型可以得到 HTTP 协议在发送者和接收者间经过的分层处理, 如图 4.1 所示, 该结构也相当于 TCP/IP 协议的模型结构。

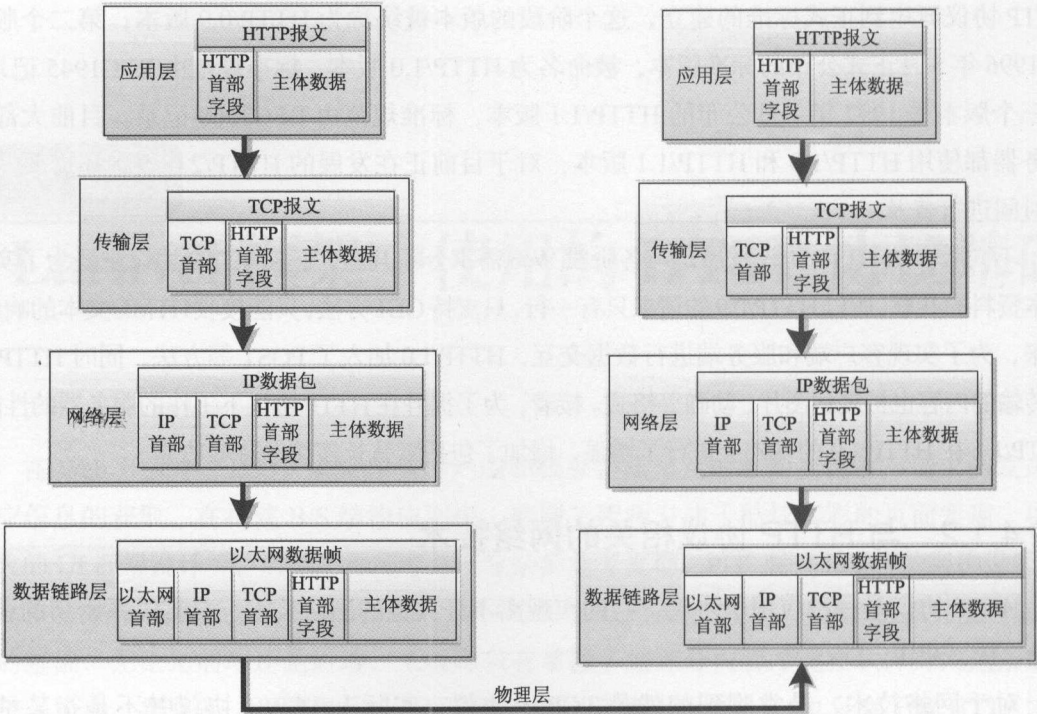


图 4.1 HTTP 协议分层结构

上面的分层结构只是一个模型，对于协议的细节没有进行定义，协议的具体定义和实现可以参看其他书籍。这里为了真实了解各层数据包的结构信息，通过 Wireshark 软件抓取浏览器访问百度网站的 HTTP 数据包进行分析，数据链路层数据包首部格式、网络层 IP 数据包首部格式、传输层 TCP 数据包首部格式、应用层 HTTP 数据包首部格式分别如图 4.2~图 4.5 所示。每一层都包含发送者和接收者目的地址及上一层协议，对于传输层，由于数据包发送到 80 端口，默认 80 端口是由服务器监听的网络端口，所以在该端口号接收的数据包都由服务器来解析。

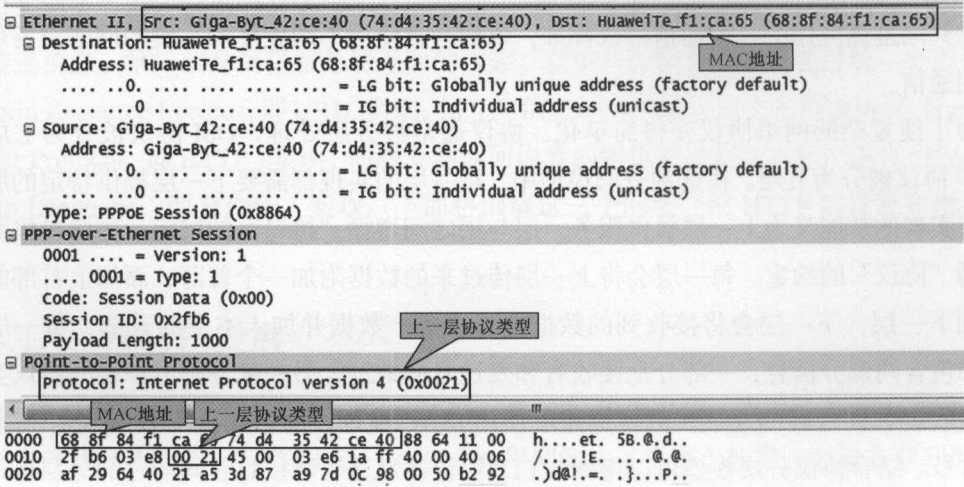


图 4.2 数据链路层数据包首部格式

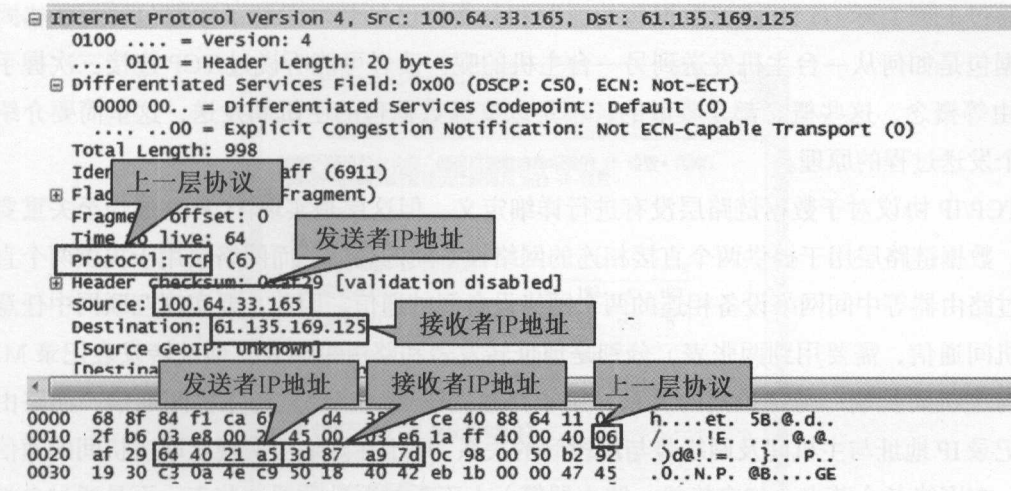


图 4.3 网络层 IP 数据包首部格式

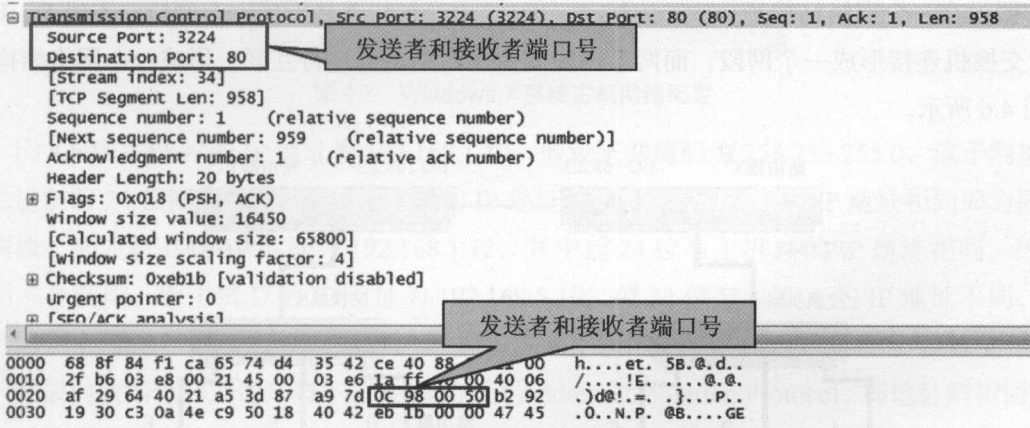


图 4.4 传输层 TCP 数据包首部格式

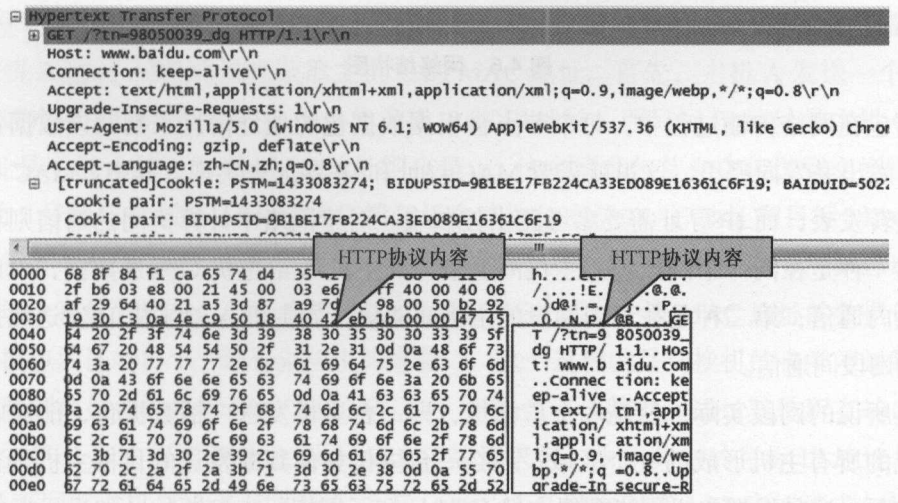


图 4.5 应用层 HTTP 数据包首部格式

通过上面实际 HTTP 协议数据包的抓取和分析，已经基本了解了数据包的封装格式，但数据包是如何从一台主机发送到另一台主机的呢？读者可能听说过 TCP 连接三次握手、IP 路由等概念，这些概念都是零散的，不足以支撑数据包的主机间发送，这里简要介绍一下整个发送过程的原理。

TCP/IP 协议对于数据链路层没有进行详细定义，但这层是实现 TCP/IP 通信至关重要的一层。数据链路层用于提供两个直接相连的网络设备间的通信，而网络层用于提供两个直接或通过路由器等中间网络设备相连的两个网络设备间的通信。因此，要实现互联网中任意两台主机间通信，需要用到两张表，分别是地址转发表和路由控制表，地址转发表记录 MAC 地址与主机间及端口与主机间的关系，用于在数据链路层实现直连主机间通信，而路由控制表记录 IP 地址与主机间及网络号与网段间的关系，用于在网络层实现互联主机间的通信。目前，在网络各个节点（如交换机、路由器等）上不需要手动设置这些表，而是通过自学习来自动生成的。下面通过一个具体实例来介绍主机间是如何实现通信的。

假设在一个网络结构拓扑中，主机 A、B、C 由交换机连接形成一个网段，主机 D、E、F 由交换机连接形成一个网段，而两个网段间由两台路由器进行互联，形成一个网络结构，如图 4.6 所示。

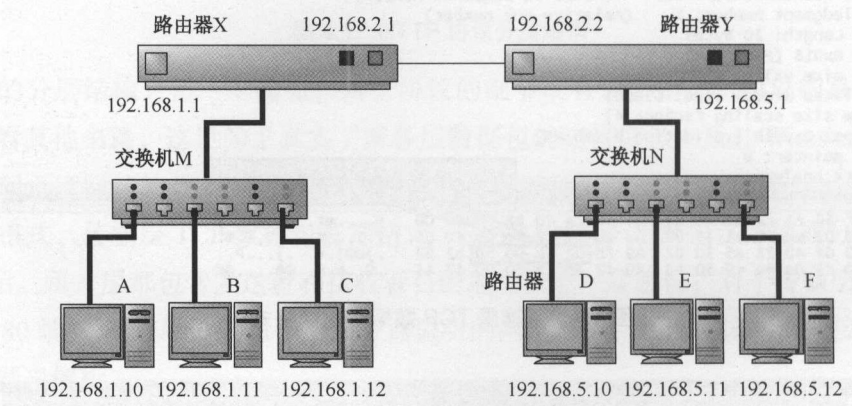


图 4.6 网络结构图

数据包在网络传输过程中，每个节点会根据数据包的地址信息来判断该数据包应该从哪个端口发出并发向哪里，这里就需要 MAC 寻址和 IP 寻址来实现。其中，MAC 寻址需要参考地址转发表，而 IP 寻址需要参考路由控制表。对于网络中计算机间的通信则存在两种情况：第一种是在同一个网段内的主机间通信，如主机 A 给主机 C 发送数据，可以简单地称为网段内通信；第二种是在不同网段的主机间通信，如主机 A 给主机 D 发送数据，可以称为不同网段间通信。

这里所说的网段实际上就是一个广播域，即一台主机发送广播数据包，能够收到该广播数据包的所有主机形成的一个区域。那么一台主机如何判断发送的目标主机是否在同一个网段内呢？这是根据主机网络配置中的 IP 地址和子网掩码来判断的，如果目标主机的 IP 地址在子网掩码的网段内，则表示发送地址为同一个网段，否则认为是不同网段的，其中

Windows 7 系统主机网络配置如图 4.7 所示。

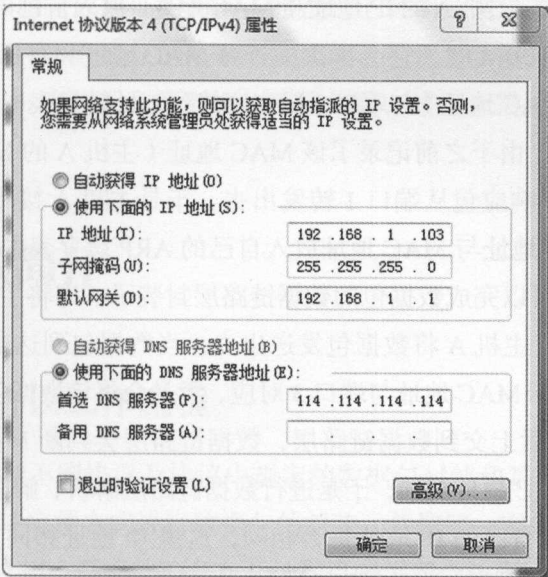


图 4.7 Windows 7 系统主机网络配置

图 4.6 中主机 A 的 IP 地址为 192.168.1.10，假设子网掩码为 255.255.255.0，该子网掩码的二进制前 24 位都为 1，如果目标主机的 IP 地址前 24 位与主机 A 的 IP 地址相同即为同一个网段，如主机 C 的 IP 地址为 192.168.1.12，其中前 24 位与主机 A 的 IP 地址相同，因此为同一个网段，而主机 D 的 IP 地址为 192.168.5.10，前 24 位与主机 A 的 IP 地址不同，则不是同一个网段。下面将分别讨论同一个网段内和不同网段间主机的数据通信过程。

网段内数据包的发送是离不开 ARP 协议（Address Resolution Protocol，即地址解析协议）的，该协议是数据链路层协议，可以利用请求包和响应包通过一台主机的 IP 地址获取它的 MAC 地址。如上面介绍的实例，主机 A 需要发送数据包到主机 C，则需要知道主机 C 的 IP 地址，但是却不知道主机 C 的 MAC 地址，因此无法完成数据包数据链路层首部的封装。此时，就需要主机 A 通过 ARP 协议来获取主机 C 的 MAC 地址。首先，主机 A 发送一个 ARP 广播请求，其中网络层首部的源 IP 地址为主机 A 的 IP 地址，而目标 IP 地址是主机 C 的，源 MAC 地址是主机 A 的，目标 MAC 地址则全是 1，这个请求包首先会传送到交换机 M，交换机 M 的 1 号端口收到该 ARP 请求广播包后会读取包中源 MAC 地址和目标 MAC 地址，假设交换机此时 MAC 地址转发表为空，则会将源 MAC 地址（即主机 A 的 MAC 地址）与对应的端口 1 存放到表中，当交换机再从其他端口接收到发往该 MAC 地址的数据包时就会直接将数据包发往端口 1，而不会发往其他端口，这个过程就是交换机的自学习过程。由于交换机查看到数据包的目标 MAC 地址全是 1，于是会将该数据包通过其他所有端口广播出去，此时主机 B 和主机 C 都会收到该 ARP 广播请求包，由于其中的目标 IP 地址与主机 B 的 IP 地址不同，主机 B 会将该数据包丢弃，而主机 C 接收到该 ARP 广播请求包后，发现与自己的 IP 地址相同，就会生成 ARP 响应数据包，数据包的源 IP 地址和源 MAC 地址为

主机 C 的 IP 地址和 MAC 地址，数据包的目的 IP 地址和目的 MAC 地址为主机 A 的 IP 地址和 MAC 地址，同时记录主机 A 的 IP 地址与 MAC 地址，放入自己的 ARP 缓存表中。此时，该 ARP 响应包通过交换机 M 时，它会再次检查源 MAC 地址和目标 MAC 地址，并将源的 MAC 地址（主机 C 的 MAC 地址）与端口 3 对应存放到地址转发表中。对于目标 MAC 地址，会首先查找地址转发表，由于之前记录了该 MAC 地址（主机 A 的 MAC 地址）与端口 1 对应，交换机只将该 ARP 响应包从端口 1 转发出去，于是主机 A 接收到该 ARP 响应包，通过解析后将主机 C 的 IP 地址与 MAC 地址放入自己的 ARP 缓存表中。此时，已经获取到了主机 C 的 MAC 地址，可以完成数据包的数据链路层封装了，即将主机 C 的 MAC 地址作为目标 MAC 地址。最后，主机 A 将数据包发送出去，当数据包到达交换机 M 时，由于地址转发表中已经记录了目标 MAC 地址与端口 3 对应，于是会将该数据包通过端口 3 转发出去，主机 C 接收到该数据包后上交到数据链路层，数据链路层会判断 MAC 地址与自己的 MAC 地址是否相同，这里 MAC 地址相同，于是进行数据链路层解析，解析完成后上交到传输层，传输层会判断 IP 地址与自己的 IP 地址是否相同，这里 IP 地址相同，于是完成传输层解析后再上交到网络层，如此一层层解析，最终完成一个网段内两台主机间数据包的传输和解析。

对于不同网段间的两台主机通信过程，这里以主机 A 与主机 D 间的通信为例，条件相同，主机 A 知道主机 D 的 IP 地址，而不知道该主机的 MAC 地址。在实现数据链路层封装时会检查主机 D 的 IP 地址是否跟自己在同一个网段（如上面介绍通过本机的 IP 地址与子网掩码判断），通过判断发现主机 D 与主机 A 不在同一个网段内，这时会用到主机网络配置的第三项——网关配置，在实际中网关的功能可能会大于路由器，即对应 TCP/IP 协议模型中传输层到应用层的数据处理部分，不仅具有路由的功能，可能还会包括协议转换、防火墙等功能，但至少要实现路由功能。对于本实例中的网关设置即为路由器 X 的端口 1 的 IP 地址，即 192.168.1.1，于是主机 A 首先将数据包发送给路由器 X 的端口 1，此时源 IP 地址和源 MAC 地址是主机 A 的，目标 IP 地址是主机 D 的，而目标 MAC 地址是路由器 X 端口 1 的，如果主机 A 目前不知道路由器 X 端口 1 的 MAC 地址，则跟网段内通信方式一样，通过 ARP 协议获取该 MAC 地址。当数据包发送到路由器 X 端口 1 时，路由器 X 会查看数据包的目标 IP 地址并查找自己的路由表，找到通往目标 IP 网段的路由，如果没有相关的路由信息，则会丢弃该数据包，当通过路由表查找到通往目标 IP 网段的发送端口为端口 2 时，于是把数据包传到端口 2 上，并由端口 2 传输到路由器 Y 端口 2 上，路由器 Y 会再次查询目标 IP 地址和自己的路由表，查找到目标 IP 所在网段的发送端口为端口 1，于是将数据包传到端口 1 并发送出去，通过交换机 N，主机 D 会接收到该数据包，从而完成主机 A 与主机 D 间的数据通信。在这期间，路由器会通过路由表查找到下一跳的 IP 地址，如果不知道 MAC 地址则通过 ARP 协议获取。路由器的路由表一般也是通过自学习获取的，这个过程的算法很多，最具代表性的有两种，即距离向量算法和链路状态算法，同时也需要 RIP、EGP 等路由协议的支持，这里就不详细介绍了，读者如果需要可以查看相关书籍。

对于以上讨论，用到了图 4.7 中主机网络配置的前三项。“DNS”是域名解析，即当通

过浏览器访问某个域名的服务器时，会首先将域名发送给 DNS 服务器，DNS 服务器将域名解析成实际 IP 地址，从而完成地址的访问。通过以上对相关网络技术的简单介绍，了解了支撑 HTTP 协议传输的 TCP/IP 协议的基本实现原理，不管是数据包封装的方式还是主机间传输的方式，在实现 Web 客户端和服务端相互通信后，接下来需要了解两者间交互内容的协议，即 HTTP 协议。

4.2 HTTP 协议简介

4.2.1 HTTP 协议工作流程

在前面已经了解多种不同协议及协议中规定的数据包封装和基本功能，HTTP 协议与其他协议一样，也需要以特定的方式进行数据包的封装，并按照一定的流程实现通信交互，而基于 HTTP 协议的请求 / 响应模式的信息交互过程可分为四个步骤。

(1) 客户端与服务器需要建立连接，如 TCP 连接。

(2) 连接建立后，客户端向服务器发送一个请求，请求报文由三部分组成：请求行、消息报头、请求内容。

(3) 服务器接到请求后，解析该请求并返回响应信息，响应报文由三部分组成：状态行、消息报头、响应内容。

(4) 客户端接收服务器所返回的信息并进行解析、处理和显示。

通常，浏览器访问一个页面，需要发出多次请求获取不同的响应内容，在 HTTP/1.0 版本中默认是传输一次数据就关闭连接，而在 HTTP/1.1 版本中默认为持久连接，即一次 TCP 连接可以完成多次 HTTP 请求。是否支持持久连接，是由消息报头中的 `connection` 字段决定的，如果请求或响应中的 `connection` 设置为 `close`，则客户端和服务器的连接为非持久连接。

4.2.2 请求报文和响应报文结构简介

HTTP 协议的请求和响应数据包一般被称为请求报文和响应报文，根据协议标准，请求报文和响应报文是由报文首部、空行和报文主体组成的，而请求报文的首部又由请求行、请求首部字段、通用首部字段、实体首部字段组成，响应报文的首部由状态行、响应首部字段、通用首部字段、实体首部字段组成。请求报文和响应报文的具体结构如图 4.8 和图 4.9 所示。

根据以上结构，可以将 HTTP 协议报文包含的内容分为五类：一是请求行，由请求方法、请求 URL、HTTP 版本组成；二是状态行，由 HTTP 版本、状态码、原因短语组成；三是首部字段，包含了请求首部、响应首部、通用首部、实体首部四种类型，一般由名字和值组成；四是其他报文首部字段，包含一些 RFC 中未定义的首部内容；五是报文主体，由任意数据组成。一个访问页面 <http://www.sina.com.cn/> 的请求报文和响应报文如图 4.10 和图 4.11 所示。

接下来逐个进行讨论。

请求行
请求首部字段
通用首部字段
实体首部字段
空行
主体

图 4.8 请求报文结构

状态行
响应首部字段
通用首部字段
实体首部字段
空行
主体

图 4.9 响应报文结构

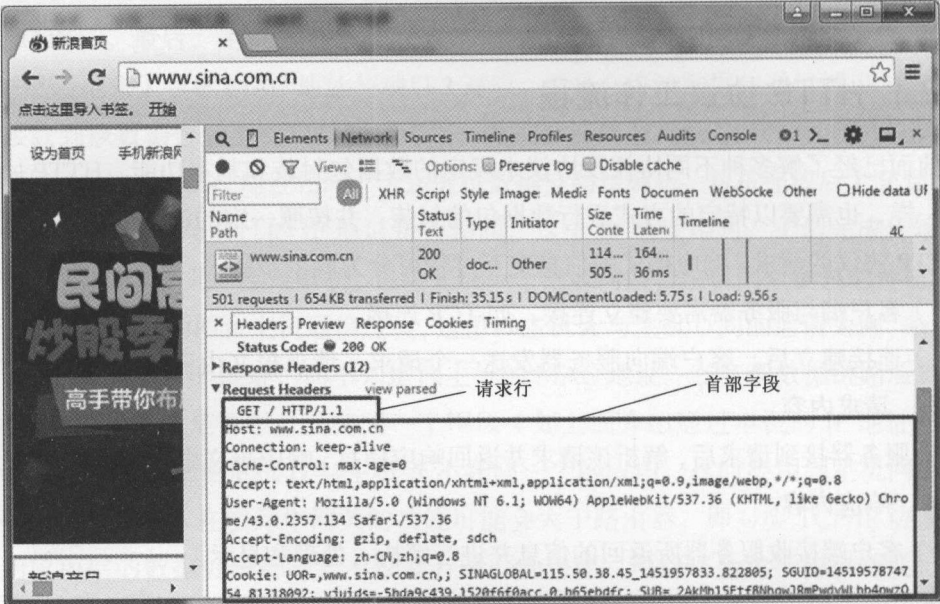


图 4.10 请求报文

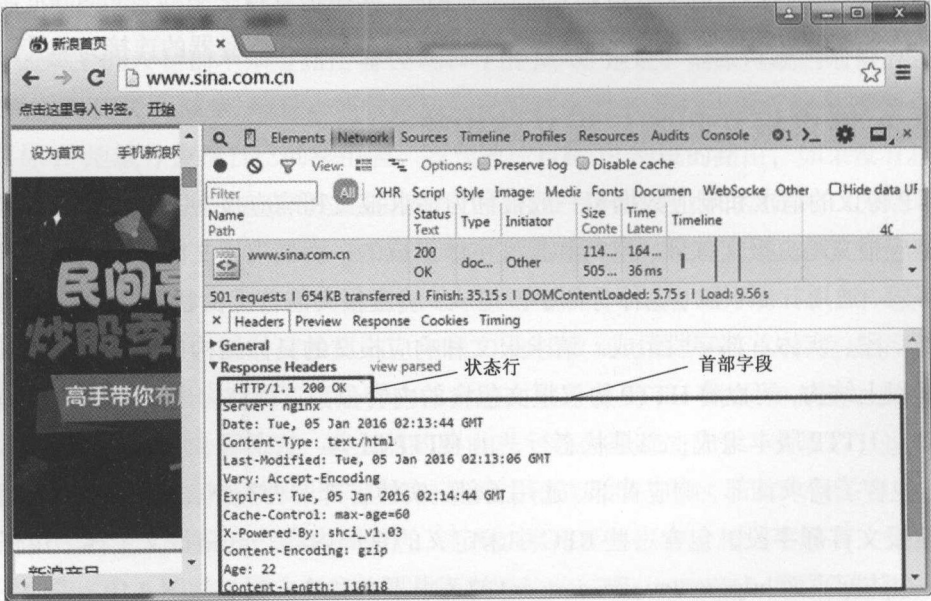


图 4.11 响应报文

(1) 请求行。

请求行中的请求方法用来描述服务器应该执行的操作，而请求 URL 指定了操作的资源，HTTP 版本用来标识客户端使用的 HTTP 版本。对于图 4.10 中访问新浪网站请求报文的请求方法为“GET”，请求 URL 为“/”，即网站根目录，HTTP 版本为“HTTP/1.1”。在 HTTP 规范中定义了一组方法，用于指定服务器的响应动作，如 GET 方法是指从服务器获取一个文本，具体请求报文方法及其功能描述如表 4.1 所示。

表 4.1 HTTP 请求报文方法及其功能

方法名称	功能描述	是否包含主体数据
GET	从服务器获取文本	否
POST	向服务发送客户端数据	是
PUT	上传客户端的文件到服务器	是
DELETE	从服务器上删除一个文件	否
HEAD	只获取服务器响应的首部	否
OPTIONS	获取服务器可以执行的方法	否
TRACE	对经过代理服务器的报文进行追踪	否

(2) 状态行。

状态行包含响应报文服务器使用的 HTTP 版本、数字状态码和原因短语，对于图 4.11 响应报文的行，其中 HTTP 版本为“HTTP/1.1”，数字状态码为“200”，原因短语为“OK”。HTTP 的状态码共分为五大类，用于描述返回的响应结果。实际 HTTP 规范中定义的状态码有几十种，比较具有代表性的状态码及含义如表 4.2 所示。

表 4.2 HTTP 响应报文状态码及含义

状态码范围	状态码类别	典型状态码	原因短语	含 义
100~199	信息类状态码	100	continue	收到请求初始部分，请继续发送
200~299	成功状态码	200	OK	请求已经被正常处理
		204	No Content	请求被成功处理，但没有主体数据
		206	Partial Content	对资源部分请求成功
300~399	重定向状态码	301	Moved Permanently	请求的 URL 资源已被更新，响应首部包含新的 URL
		302	Found	请求的 URL 资源临时更新
		304	Not Modified	请求不符合条件，需要更改条件

续表

状态码范围	状态码类别	典型状态码	原因短语	含 义
400~499	客户端错误状态码	400	Bad Request	客户端发送一个错误请求
		401	Unauthorized	请求需要包含通过 HTTP 认证的信息
		403	Forbidden	访问被拒绝
		404	Not Found	没有找到请求的 URL 资源
500~599	服务器错误状态码	500	Internal Server Error	服务器出现错误，无法提供请求的资源
		503	Service Unavailable	服务器正忙，无法提供正常请求服务

（3）首部字段。

首部字段是构成 HTTP 报文的基本要素之一，这部分包含的信息也最为丰富，主要用于规定客户端和服务端在处理请求和响应时的操作。通常，根据首部字段的用途可以分为四类，分别是通用首部字段、请求首部字段、响应首部字段和实体首部字段。首部字段通常由字段名和字段值构成，中间通过“:”分隔。下面分别简要介绍四类字段的作用和部分首部字段的含义。

• 通用首部字段

通用首部字段一般用来提供 HTTP 报文的最基本信息，这些信息无论是请求报文还是响应报文都可以使用，主要用于描述 HTTP 协议本身。比如描述 HTTP 是否持久连接的 Connection 字段、HTTP 发送日期的 Date 字段、用于缓存控制的 Cache-Control 字段等。下面列出常用的通用首部字段及含义，如表 4.3 所示。

表 4.3 HTTP 报文通用首部字段及含义

通用首部字段名	功能说明
Connection	用于指定客户端 / 服务器间连接的选项，例如指定连续连接或者通过“close”选项通知服务器在响应完成后关闭连接
Date	报文创建的日期
Transfer-Encoding	指定报文主体数据的编码方式
Update	发送端协议准备升级为新版本
Via	报文经过中间节点（网关、代理等）的信息
Cache-Control	指定缓存行为
Warning	错误通知

• 请求首部字段

请求首部字段一般用于 HTTP 请求报文中，主要描述客户端的版本、处理能力等信息，用于帮助服务器更好地提供请求响应的字段，如告诉服务器只接收某种响应内容的 Accept 字段、发送 Cookies 的 Cookie 字段等。下面列出常用的请求首部字段及含义，如表 4.4 所示。

表 4.4 HTTP 报文请求首部字段及含义

请求首部字段名	功能说明
Accept	客户端可处理的媒体类型
Accept-Charset	客户端优先接收的字符集
Accept-Encodiing	客户端优先接收的编码方式，如 gzip
Accept-Language	客户端希望的语言种类
Authorization	授权信息，通常用于对服务器发送的 WWW-Authenticate 首部进行应答
Cookie	用于客户端向服务器发送一个令牌
Cookie2	用来说明客户端支持的 Cookie 版本
From	客户端的 E-mail 地址，由一些特殊的 Web 客户程序使用，浏览器不会用到它
Host	请求资源所在服务器的主机和端口
If-Modified-Since	只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回 304 “Not Modified” 应答
Referer	包含一个 URL，客户端从该 URL 页面出发访问请求页面
User-Agent	客户端信息，如客户端应用程序名称、版本等
UA-OS	客户端主机上的操作系统名称、版本等信息
UA-Pixels	客户端显示器像素信息

• 响应首部字段

响应首部字段是描述 HTTP 响应本身的字段，一般用于 HTTP 响应报文中，主要用于描述响应主机的信息、功能等，通过响应首部字段有助于客户端处理响应，并在将来发出更适合服务器的请求，如定时刷新的 Refresh 头、设置 Cookie 的 Set-Cookie 头等。下面列出常用的响应首部字段及含义，如表 4.5 所示。

表 4.5 HTTP 报文响应首部字段及含义

响应首部字段名	功能含义
Age	响应资源创建的时间
Set-Cookie	在客户端设置一个 Cookie，服务器用来对客户端进行标识
Server	服务器应用程序软件的名称、版本等信息
Retry-After	当资源不可用时，在此期间再次发送请求
Refresh	表示浏览器应该在多长时间之后刷新文档，以秒计

• 实体首部字段

实体首部字段用于描述 HTTP 报文主体数据的字段。由于报文主体可以存在于请求报文和响应报文中，所以这种类型首部字段可以应用在这两种类型的报文中。实体首部字段用于描述主体内容的元信息，包括实体信息类型、长度、压缩方法、最后一次修改时间、数据有效性等，以便告诉接收者如何进行处理，如 Content-Language 用于描述报文主体的语言等。

下面列出常用的实体首部字段及含义，如表 4.6 所示。

表 4.6 HTTP 报文实体首部字段及含义

实体首部字段名	功 能 含 义
Allow	资源主体可以支持的 HTTP 请求方法
Content-Location	通知接收端主体的 URL，用于接收端定位到该资源
Content-Encoding	主体适用的编码方式
Content-Language	解释主体时适用的语言
Content-Length	主体的大小，通常以字节数表示
Content-MD5	主体的 MD5 校验和
Content-Type	主体的对象类型

HTTP 协议作为应用层协议离不开 TCP/IP 协议的支持，本章从网络协议开始，对 HTTP 协议进行了简单的介绍，要想了解 Laravel 框架的方方面面，那么这些底层的知识是必须要掌握的，只有了解这些才能对 Laravel 框架底层实现有更深刻的认识。对于 Web 应用无非是获取请求、返回响应，而请求的获取与响应的返回又需要满足 HTTP 协议才能正常工作，Laravel 框架的底层使用了 Symfony 的模块用于处理 HTTP 请求和响应，可以看出 Laravel 框架在构建过程中也大量使用了已有的模块，这也将是未来软件开发的趋势。

第5章

Laravel 框架初识

通过前面几章基础内容的学习，对于了解 Laravel 框架的基本功能已经足够了，本章将对 Laravel 框架的基本内容进行讲解，主要包括程序的组织结构和三个重要环节，即路由、控制器和视图。

5.1 Laravel 框架应用程序目录结构

默认的 Laravel 应用程序框架按照 PSR-0 和 PSR-4 规范组织文件的目录结构，通过这种规范的结构，可以更好地实现文件的自动加载和分类。当然，读者也可以按照自己的要求自由地组织 Laravel 框架的结构，但要注意，自己组织的目录结构或者新添加的文件需要 Composer 实现自动加载。

5.1.1 Laravel 框架应用程序根目录介绍

前面提到，默认的 Laravel 框架应用程序是符合 PSR 规范的，所以相应的目录结构也是基本固定的，不同目录加载了不同的功能文件，如果添加了新的目录，需要在 `composer.json` 文件中添加 PSR 规范的自动加载部分并执行 `update` 命令，否则将无法实现文件的自动加载。Laravel 框架根目录组织结构如下。

app: 主要包含应用程序的核心代码，用户构建应用的大部分工作都在这个目录下进行，包括路由文件、控制器文件、模型文件等。

bootstrap: 主要包含几个框架启动和自动加载配置的文件。

config: 主要包含应用程序常用的配置文件信息。

database: 主要包含数据库迁移和数据填充文件。

public: 为应用程序的入口目录，包含应用程序入口文件 `index.php`，同时包含静态资源文件如 CSS、JavaScript、images 等。

resources: 主要包含视图文件。

storage: 包含编译后的 Blade 模板、基于文件的 session、文件缓存和日志等文件。

tests：主要包含自动化测试文件。

vendor：主要包含依赖库文件，其中包括 Laravel 框架源代码。

.env 文件：一个重要的文件，为 Laravel 框架主配置文件。

composer.json 文件：composer 项目依赖管理文件。

5.1.2 app 目录介绍

应用程序的大部分内容都存在于 app 目录下，该目录同样通过 composer 使用自动加载标准（PSR-4）来自动加载其中的文件。如果想要改变这个目录下的命名空间，可以通过 artisan 命令 app:name 完成，如 `php artisan app:name Application`，也可以直接修改 composer.json 文件来实现。在使用 Laravel 框架开发项目时，几乎大部分的新开发功能也会放置在这个目录下，下面介绍该目录下的组织结构。

Console：主要包含所有的 artisan 命令。

Events：用来放置与事件相关的类。

Exceptions：包含应用程序的异常处理类，用于处理应用程序抛出的任何异常。

Http：主要包含路由文件、控制器文件、请求文件、中间文件等，是应用程序与 Laravel 框架源代码等外部库交互的主要地方。

Jobs：主要包含消息队列的各种消息类文件。

Listeners：主要包含监听事件类文件。

Providers：主要包含服务提供者的相关文件。

app 目录下可以放置模型类文件，用于操作数据，如默认的 User.php 文件。

需要注意的是，由于 app 目录已经通过 composer 包含到自动加载目录中，所以在此目录下新建目录时不需要再更新自动加载类就可以索引到，但是类的命名空间需要与文件目录相符。同时，app 目录中的许多类可以用 artisan 命令产生。要查看可以使用哪些命令，可以在 Laravel 根目录下执行 `php artisan list make` 命令获取。

5.1.3 vendor 目录介绍

vendor 目录主要包含 Laravel 应用程序的外部依赖库，包括 Laravel 框架源代码部分。该目录中文件的组织结构是根据依赖关系决定的，每一个文件夹都是一个功能模块，可以单独通过 composer 下载该组件进行使用，对于像 Laravel 框架这样的大组件还会包含很多的小组件，相当于整个 vendor 目录就是由一个个相互依赖的功能组件模块组织起来的，它们可以独立工作，也可以被组织起来协调工作。下面介绍几个主要的目录结构。

composer：主要包含 composer 按照 PSR 规范生成的自动加载类。应用程序类的自动加载都是由这部分实现的。

laravel：包含 Laravel 框架源代码，代码部分都包含在 `vendor\laravel\framework\src\`

Illuminate 文件夹下，在该文件夹下又包含很多文件夹，每一个文件夹又是一个组件，如用于管理 session 的 session 组件、用于实现路由功能的 routing 组件，这些组件都是可以独立工作的。

symfony: Laravel 框架的底层（如请求类、响应类、文件管理类等）使用了 symfony 框架的部分，所以该目录包含这部分内容。

monolog: 包括日志记录模块文件。

phpunit: 包含程序单元测试模块文件。

5.2 Laravel 框架应用程序的三个重要环节

根据 HTTP 协议，服务端接收到客户端的请求后，会根据请求的统一资源定位符（URL）进行分发处理，找到对应的响应处理程序，从而返回对应的响应。在 Laravel 框架应用程序中，根据请求返回响应可以分为三个阶段，即路由阶段、控制器阶段和视图阶段。当然，也可以只有路由阶段和控制器阶段。下面将简单介绍这三部分的内容，这三部分可以算是请求到响应生成的三个环节，Laravel 框架程序的整个流程将会在第 7 章中介绍。这部分内容在 Laravel 官网上基本都可以找到，这里只是进行了重新编排和详细介绍，使读者更容易理解。如果需要，读者可以在官网进行查阅。

5.2.1 路由

路由在 Laravel 框架中的作用是根据请求资源定位符的不同，将用户的请求按照事先规划的方案提交给指定的控制器或者功能函数来进行处理。

Laravel 框架应用程序的路由大部分记录在 `laravel/app/Http/routes.php` 文件中，该文件在应用程序路由服务提供者启动过程中，通过 `app/Providers/RouteServiceProvider.php` 文件中的 `map()` 方法和代码“`require app_path('Http/routes.php');`”进行加载，这部分内容将在第 7 章中详细介绍，这里只介绍不同的路由设置方案，让读者对 Laravel 框架应用程序请求响应过程中的三个主要部分有个初步认识。

1. 基础路由设置

Laravel 框架中的基本路由是一个资源定位符（URL）对应一个响应程序，这个程序可以是一个闭包函数，也可以是一个控制器响应函数的标识，具体格式和实例如下。

路由定义格式：Route::方法名('资源标识', 闭包函数或控制器响应函数标识)

具体实例如下（实例中设置的请求主机地址为 `www.laravelstudy.com`）：

```
Route::get('/', function(){ return 'Hello Laravel';});
```

该实例定义了一个路由，将请求网站地址的根目录路由到一个处理函数（该函数为一

个闭包函数)，处理函数的返回值就作为请求的响应，这里的响应就是输出一条“Hello Laravel”语句。下面介绍一个其他 URL 的路由，实例代码如下：

```
Route::get('home', function(){return 'Hello Laravel';});
```

该实例将地址为 `www.laravelstudy.com/home` 的 GET 请求路由到处理函数。如果请求为其他方法，则需要使用对应的路由设置方法进行定义，对应的方法有 `get`、`post`、`put`、`delete` 等。对于多种请求的路由定义可以通过 `match` 和 `any` 方法实现，`match` 需要设置请求方法数组、URL 地址和请求处理函数三个参数，第一个参数要以数组的方式设置允许的请求方法，而 `any` 方法则对应所有 HTTP 请求方法。实例代码如下：

```
Route::match(['get', 'post'], '/', function(){ return 'Hello Laravel';
});
Route::any('home', function(){ return 'Hello Laravel';});
```

2. 路由参数

通常情况下，GET 请求可以通过 URL 地址（QueryString）的方式向服务器传递参数，在 Laravel 框架中有更加优雅的形式来传递参数，即通过路由参数进行传递。路由参数主要针对的是 GET 请求的情况。通过路由参数传递参数的具体格式如下：

```
Route::get('资源标识/{参数名[?]}[{参数名}...] ', 闭包函数或控制器响应函数标识)
[->where('参数名', '正则表达式')];
```

通过这种路由形式传递的参数，闭包函数或控制器函数可以直接在路由处理函数中添加名称为“\$ 参数名”的形参实现对路由参数的接收。[] 中的内容是可选择添加的，[?] 是指路由参数，也可选择添加，[->where('参数名', '正则表达式')] 用来对输入参数进行限制，只有符合正则表达式要求的参数才可以传递。对于多个参数，可以用关联数组的形式定义限制条件。具体实例如下：

```
Route::get('user/{id}',function($id){
    return '$id='.$id;
});
```

对于上述实例，当请求地址为 `www.laravelstudy.com/user/3` 的 GET 请求时，路由参数 3 会传给闭包函数的参数 `$id`，此时输出响应 `$id=3`。此时，路由参数必须要添加，否则将无法找到匹配的路由信息，也就无法找到对应的响应函数，结果是无法对请求做出响应。

```
Route::get('user/{name?}',function($name=NULL){
    return '$name='.$name;
});
```

此时定义了可选路由参数，当请求地址为 `www.laravelstudy.com/user/xiaoming` 的 GET 请求时，回调函数的参数“\$name”会接收参数值“xiaoming”。此时参数是可选的，当参数存在时，函数参数会正常接收，如果不存在，则会用默认值。下面给出一个对路由参数进行验证限制的实例：

```
Route::get('user/{name}', function($name)
{
    return '$name'.'$name';
})->where('name', '[A-Za-z]+');
```

此时对路由参数添加了正则表达式验证（'[A-Za-z]+'，匹配由字母组成且不少于1个字母的名字），只有满足正则表达式的路由参数才能匹配此路由，否则无法找到对应路由。下面定义多个路由参数，并由数组形式对路由参数进行验证。

```
Route::get('user/{id}/{name}', function($id, $name)
{
    return '$id='.'$id.'and'.'$name='.'$name';
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

3. 路由命名

路由命名相当于在路由定义的时候，为路由起了一个别名，在以后的程序中可以通过这个别名来获取路由的信息。实例如下：

```
Route::get('user/name', ['as' => 'name', 'uses'=>function()
{
    return "Hello Laravel";
}]);
```

这里为路由起了一个别名，以后可以使用这个路由名称产生 URL 地址或进行重定向，如“\$url = route('name');”获取的路由地址为 www.laravelstudy.com/user/name。

4. 路由群组

当一个项目很大时，会定义很多的路由，对路由进行分组将使得程序更加规范易读，而路由群组就是给某一类路由进行分组，同时给这个路由组添加中间件、前缀、子域名等，使得路由定义更加简洁。假设需要定义多个关于用户管理的路由，通过普通路由定义方式如下：

```
Route::get('user/id', ['middleware' => 'auth', function() {
    return "Hello Laravel";
}]);
Route::get('user/name', ['middleware' => 'auth', function() {
    return "Hello Laravel";
}]);
```

这里定义了两条路由信息，其中前缀都为 user，而中间件都为 auth，这种情况就可以将其定义为一个路由群组。

```
Route::group(['prefix' => 'user', 'middleware' => 'auth'], function() {
    Route::get('id', function() {
        return "Hello Laravel";
    });
});
```

```

    });
    Route::get('name', function() {
        return "Hello Laravel";
    });
});

```

通过以上对路由设置方法的介绍，读者对 Laravel 框架中路由的概念有所了解，目前我们只是看到了 Laravel 框架应用程序中的一个环节，还无法理解这个环节在整个流程中的作用。这里可以将路由设置的过程理解为在程序内部定义了一个路由表，与网络中使用的路由器一样，也事先具有一张路由表，当不同的请求到来时，根据路由表选择不同的处理程序，如同具有不同 IP 地址的数据包经过路由器将被传递到不同的线路上一样。请求经过路由表路由后，大部分还会经过中间件，再到控制器，一部分又会经过视图处理得到响应。下面继续介绍剩下的两个重要环节——控制器和视图。

5.2.2 控制器

前面介绍了路由，路由中定义了一定的请求分发逻辑，如果只用这部分对于简单的 Web 应用是可以的，但当程序的规模扩大后，程序逻辑将变得更加复杂，所以分层的逻辑处理更加适合，这时就用到了第二层分发逻辑处理单元——控制器。

1. 控制器简介

Laravel 框架程序与其他框架类似，控制器也是由类来组织的，通过不同的类可以将 HTTP 请求按照内部关系进行分类，同一类的请求可以放在一个控制器中来处理。控制器类通常放在 `laravel/app/Http/Controllers` 目录下。这里给出一个简单的控制器类的代码：

文件：`laravel\app\Http\Controllers\HomeController.php`

```

<?php namespace App\Http\Controllers;
use Illuminate\Routing\Controller as BaseController;
class HomeController extends BaseController
{
    public function index($username)
    {
        return 'Hello'. $username;
    }
}

```

可以看到，Laravel 框架应用程序的控制器非常简单，只要定义一个控制器名称，并继承基础控制类 `App\Http\Controllers\Controller` 就可以生成一个新的控制器，该基础控制器类继承自 `Illuminate\Routing\Controller` 类，其中定义了控制器所需要的基本方法。控制器类作为 HTTP 请求的二次分发控制部分，与路由有着紧密的关系，然而，Laravel 框架通过服务容器，以依赖注入的方式解决了这种耦合关系，使得控制器类与其他类的耦合度非常低，这应该是

Laravel 框架设计之所以艺术的地方，这部分内容将在后续章节进一步介绍。

2. 控制器路由

前面我们讲解路由的时候，处理函数是由闭包函数完成的，其实路由也可以将 HTTP 请求分发到控制器中的函数进行处理，这就是控制器路由。控制器路由有三种，分别是基础控制器路由、隐式控制器路由和 RESTful 资源控制器路由，其实后两种只是用一条控制器路由命令解决多条 HTTP 请求分发的问题，这就需要控制器按照固定的模式进行设置，下面对上述内容逐一进行介绍。

(1) 基础控制器路由。

基础控制器路由的设置和前面讲述的路由设置格式基本相同，只是将闭包函数换成“控制器类名 @ 函数名”格式的字符串，通过“@”符号将控制器类名和函数名进行分隔，这样我们就唯一定位了一个 HTTP 请求的响应函数。基础控制器路由通过“Route:: 请求方法”定义，格式如下：

```
Route:: 请求方法 ('资源标识 / { 参数名 [?] [/ { 参数名 } ...] ', '控制器类名 @ 函数名称');
```

基础控制器路由的请求方法和 URL 地址与前文中路由的介绍相同，而处理函数前文中使用的是闭包函数，这里使用一个字符串来定义处理函数，该字符串包含两部分内容，即控制器类名称和函数名称，两个名称中间用“@”符号隔开。下面给出基础控制器路由的实例：

```
Route::get('home/{ name }', 'HomeController@index');
```

该控制器路由定义了对于 URL 地址为 www.laravelstudy.com/home/xiaoming 的 GET 请求将由 HomeController 控制器中的 index(\$username) 函数进行解析。控制器路由同样可以传递路由参数，参数的接收由控制器中的函数形参完成，接收方式是按路由参数（{ name }）和控制器函数形参（\$username）的顺序进行接收，与名字无关。如果存在可选择添加路由参数，则需要将其放在路由参数之后，否则会出现 HTTP 请求无法找到路由的错误。具体实例如下：

控制器路由：

```
Route::get('home/{id}/{name?}', 'HomeController@index');
```

控制器：

```
<?php namespace App\Http\Controllers;
use Illuminate\Routing\Controller as BaseController;
class HomeController extends BaseController
{
    public function index($name,$id=null)
    {
        return 'Hello, '.$name.', '.$id;
    }
}
```

当通过浏览器访问地址 `www.laravelstudy.com/home/12/wshuo` 时，浏览器返回响应 “Hello,12,wshuo”。通过该实例可以看出控制器路由中的 `{id}` 由控制器函数中的 `$name` 参数接收，而 `{name?}` 由 `$id` 参数接收，说明接收方式与顺序有关，与名称无关，且可选路由参数放在路由参数之后。

（2）隐式控制器路由。

基础控制器路由需要对路由进行逐个定义，这种定义结构清晰灵活但效率很低，而隐式控制器路由是一条语句定义多条路由信息。隐式控制器路由通过 `Route::controller` 方法定义，具体格式如下：

```
Route::controller('路由前缀', '控制器类名' [, 命名路由]);
```

在该方法中，可以提供三个参数，其中前两个参数是必须的，最后一个参数是当需要添加命名路由时才以关联数组的形式给出，一般情况下，使用前两个参数。第一个参数表示路由前缀，第二个参数表示控制器类名。实例如下：

隐式控制器路由：

```
Route::controller('home', 'HomeController');
```

控制器类：

```
<?php namespace App\Http\Controllers;
use Illuminate\Routing\Controller as BaseController;
class HomeController extends BaseController
{
    public function getIndex($username)
    {
        return 'Hello' . $username;
    }
}
```

通过设置如上形式的隐式控制器路由，当客户端发出 URL 地址为 `www.laravelstudy.com/home/index/xiaoming` 的 GET 请求时 (`xiaoming` 为路由参数)，控制器 `HomeController` 类中的 `getIndex($username)` 函数将对请求进行处理，即请求地址为“主机地址 / 路由前缀 / 控制器方法名 / 路由参数”结构时，控制类中的处理函数名为请求方法加上控制器方法名，其中控制器方法名首字母要大写。如果请求方法不加限制，也可以用 `anyIndex($username)` 这样的方法名。这里需要注意两点，一是路由前缀可以为“/”，二是控制器方法名为多个单词（如 `getHomeIndex`）时，请求地址不能是 `homeIndex` 的形式，而应该是 `home-index` 的形式，否则无法对请求正确路由。

（3）RESTful 资源控制器路由。

RESTful 是一种接口设计的思想和规范，这个思想最早由 Roy Thomas Fielding 在 2000 年提出，REST 是英文 `Representational State Transfer` 的缩写，这里我们暂且直译为“表现状态转化”。Laravel 的 RESTful 资源控制器路由支持了这种接口规范。在实际应用中往往路

由的逻辑较为复杂，RESTFul 资源控制器路由无法满足这种复杂应用，但作为框架接口的一种设计思想是比较有借鉴意义的。具体定义格式如下：

```
Route::resource(' 根资源标识 ', ' 控制器类名 ');
```

通过此语句可以创建多条路由，用来处理不同的 HTTP 请求行为，控制器类中请求处理方法与 HTTP 请求的对应关系如表 5.1 所示，下面给出相应的实例。

表 5.1 RESTFul 接口请求方法与控制器处理函数对应表

请 求 方 法	路 径	行 为	控制器类处理函数名
GET	/home	索引	index
GET	/home/create	创建	create
POST	/home	保存	store
GET	/home/{id}	显示	show
GET	/home/{id}/edit	编辑	edit
PUT/PATCH	/home/{id}	更新	update
DELETE	/home/{id}	删除	destroy

RESTFul 资源控制器路由：

```
Route::resource('home', 'HomeController');
```

控制器类：

```
<?php namespace App\Http\Controllers;
use Illuminate\Routing\Controller as BaseController;
class HomeController extends BaseController
{
    public function index()
    {
        return 'index';
    }
    public function create()
    {
        return 'create';
    }
    public function store()
    {
        return 'store';
    }
    public function show($id)
    {
        return 'show' . $id;
    }
    public function edit($id)
    {
```



```

        return 'edit'.'.$id;
    }
    public function update($id)
    {
        return 'update'.'.$id;
    }
    public function destroy($id)
    {
        return 'destroy'.'.$id;
    }
}

```

根据以上定义的 RESTful 资源控制器路由，当发送地址为 `www.laravelstudy.com/home` 的 GET 请求时，HomeController 控制器类中的 `index` 方法将会处理请求。当发送地址为 `www.laravelstudy.com/home/create` 的 GET 请求时，就会由 `create` 方法进行处理。当发送地址为 `www.laravelstudy.com/home` 的 POST 请求时，将会由 `store` 方法处理。

5.2.3 视图

视图用于向用户呈现网页界面，一个文件只要向客户端输出可视内容，都称为一个视图。Laravel 框架应用程序将视图作为一个独立的组件与控制器解耦，所以在任何位置都可以使用 `view()` 的方式加载一个视图。Laravel 框架最吸引人的地方就是这些看似紧密联系的环节却又松耦合，使得各个部分添加、修改代码可以很小地影响其他部分，从而可以独立开发、任意组合。

1. 基本用法

Laravel 框架有两种方式生成视图，一种是以字符串的形式返回视图文件源代码，另一种是返回 Laravel 框架中的视图文件。第一种在前面的路由和控制器部分已经用到，通常以“`return '字符串';`”的形式实现，对于较为复杂的字符串可以使用 Heredoc 结构，实例如下：

```

public function index()
{
    return <<<STR
    <!doctype html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Laravel</title>
    </head>
    <body>
        //content
    </body>

```

```

</html>
STR;
}

```

在 Laravel 框架中，视图文件是以“.blade.php”后缀命名的，当然对于“.php”命名的文件也可以解析，解析时可以使用 `view()` 方法返回视图响应。通常，视图文件保存在 `laravel/resources/views/` 文件夹内。实际上，`view()` 函数是在文件 `Illuminate/Foundation/helpers.php` 中定义的全局函数，其实相当于 `View::make()` 方法，最终调用的是 `Illuminate\View\Factory` 中的 `make()` 方法，进而创建 `Illuminate\View\View` 类的实例化对象。至于其中的细节，将在后续章节中进行介绍。这里给出一个简单的实例：

```

public function index()
{
    return view('index');
}

```

默认情况下，上述实例会加载 `laravel/resources/views/index.blade.php` 文件，如果视图文件是存放在 `laravel/resources/views/` 的子文件夹内，比如，如果视图文件是 `laravel/resources/views/user/login.blade.php`，则可以通过以下代码返回视图文件。

```

public function index()
{
    return view('user.login');
}

```

同时，也可以以更加直观的文件夹格式返回视图文件。

```

public function index()
{
    return view('user/login');
}

```

2. 数据传递

在实际应用中，很多时候我们需要将数据传递到视图中用于显示，如我们在登录一个网站后要显示用户名。一般有三种方式可以实现数据的传递，方式一是通过数组形式；方式二是通过 `with()` 函数形式；方式三是通过 `with` 加变量名的形式，这种方法也被称为魔术方法。假设我们需要将以下两个变量传递到 `index` 视图文件中，数据内容如下：

```
$username = xiaoming    $age = 18
```

通过数组方式：

```

public function index()
{
    return view('index', ['username'=>'xiaoming', 'age'=>18]);
}

```

```
}
```

直接将一个数组以第二参数的形式传递给 view() 函数：

```
public function index()
{
    $data = ['username'=>'xiaoming', 'age'=>18];
    return view('index',$data);
}
```

通过 with() 函数方式：

```
public function index()
{
    return view('index')->with('username','xiaoming')->with('age',18);
}
```

通过 with 加变量名方式：

```
public function index()
{
    return view('index')->withUsername('xiaoming')->withAge(18);
}
```

通过上述实例，我们看到三种方式都可以向视图中传递数据，但是以数组的方式最为简便优美，一般情况我们使用这种方式来实现数据传递。

3. blade 模板

Blade 模板是 Laravel 所提供的视图文件模板引擎，该模板引擎通过模板继承和区块可以实现高度的代码复用和清晰的视图结构。以 blade 模板定义的视图文件需要在文件名称后加 “.blade.php” 后缀名。

(1) blade 模板结构布局标签。

通常情况下，有些视图的文件头、侧边栏、页脚及加载的外部文件都是相同的，如果要在各个页面都重复添加将会非常麻烦，通过 blade 模板，我们可以定义页面布局文件，而其他文件使用这个布局文件即可。下面使用 blade 模板引擎定义一个视图文件。页面布局文件 layout.blade.php 代码如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title> 思维笔记 </title>
        <link href="/backyard/css/bootstrap.min.css" rel="stylesheet">
    </head>
```



```

<body>
    @section('navbar')
        This is the initial navbar.
    @show
        @yield('content')
    <footer>
        <p>&copy; Company 2015</p>
    </footer>
    <script src="/backyard/js/jquery.min.js"></script>
</body>
</html>

```

下面通过页面布局文件创建一个视图文件 `view.blade.php`，代码如下：

```

@extends('layout')
@section('navbar')
    @parent
    <p>This is the appended navbar.</p>
@endsection
@section('content')
<div class="container">
    this is the content!!!
    @include('sidebar')
</div>
@endsection

```

其中，子视图文件 `sidebar.blade.php` 内容如下：

```

<div class="sidebar" id="sidebar">
    this is the sidebar!!!
</div>

```

最终，通过 `return view('view')` 语句输出视图显示内容：

```

This is the initial navbar.
This is the appended navbar.
this is the content!!!
this is the sidebar!!!
© Company 2015

```

通过以上视图文件的设计，可以看到通过 `blade` 模板可以定义布局文件，将基本不变的部分都在布局文件中进行定义，而将改变的部分定义为区块。视图文件通过继承布局文件并覆盖需要改变的区块，加载子视图文件，最终实现视图文件的组装。使用 `blade` 模板设计的视图文件结构清晰、代码高度复用、语法简单。下面介绍 `blade` 模板中用于布局的语法标签。

`@extend('布局文件名')`：用于继承一个布局文件。

`@section(' 区块名 ')`: 用于定义一个区块, 它可以有不同的结尾标识, `@show` 用于显示这个区块, `@stop` 和 `@endsection` 用于结束一个区块, `@overwrite` 用于重写前面的区块。其中, 如果在布局模板文件中用 `@stop` 或 `@endsection` 结束这个区块, 则视图文件将无法覆盖这个区块。

`@parent`: 用于显示继承的布局模板中的内容。

`@yield(' 区块文件 ', ' 默认内容 ')`: 用于在布局文件中定义一个区块, 在视图文件中同样可以通过 `@section` 和 `@stop` 或 `@endsection` 来定义这个区块, 如果视图文件没有定义这个区块, 将以默认内容输出。其中, 默认内容不是必须的, 可以不添加。

`@include(' 子视图名称 ')`: 用于在视图文件中加载子视图文件, 使得视图文件结构清晰。

(2) blade 模板过程控制语法标签。

Blade 模板定义了很多语法标签, 前面介绍了布局的语法标签, 下面介绍简化视图文件设计的过程控制语法, 主要包括变量输出和控制流程语法。下面给出变量输出的实例。

变量输出格式:

```
{{ 变量或返回值为一个变量的函数 }}
```

原生 PHP 代码实现:

```
<?php echo $name; ?>
```

blade 模板代码实现:

```
{{ $name }}
```

下面给出检查数据是否存在并输出的实例。

原生 PHP 代码:

```
<?php echo isset($name) ? $name : 'Default';?>
```

blade 模板代码:

```
{{ $name or 'default' }}
```

如果想输出有双花括号包起来的内容, 则可以在花括号之前加上 `@` 标识来禁止 blade 模板解析, 如 `@{{ string }}`。下面给出控制流程 if 语句的实例。

原生 PHP 代码:

```
<?php
if($username == 'xiaoming'){
    echo "I am xiaoming";
}else{
    echo "I am xiaofang";
}
```

```
?>
```

blade 模板代码:

```
@if($username=='xiaoming')
    i am {{$username}}
@else
    i am xiaofang
@endif
```

下面给出控制流程 foreach 循环的实例。

原生 PHP 代码:

```
<?php
foreach ($users as $user) {
    echo '<p>'. "this user is $user". '</p>';
}
?>
```

blade 模板代码:

```
@foreach ($users as $user)
    <p>This user is {{$user}}</p>
@endforeach
```

除此循环控制外, for、while 循环使用方法也是相同的, 这里不再赘述, 读者可以在官方文档中找到相关内容。

本章介绍了 Laravel 框架的基本概况, 包括路由、控制器和视图, 这三个模块也是使用该框架构建项目时需要重点设计实现的部分, 对于大部分初学者来说首先接触的也是这三个方面, 它们呈现了框架的外貌, 通过修改这些部分就可以搭建一个简单的 Web 应用, 但是只知道这些还远远不够, 要想很好地驾驭该框架, 不仅需要了解怎样使用它, 还需要了解它工作背后的原理, 只有知其然并知其所以然才能在构建项目时得心应手、灵活使用, 而这些背后的机制将在后续章节逐步介绍。

第6章

Laravel 框架中的设计模式

再高级的东西最多也就那么几个部分高级，就算人也只是拥有高级的大脑。在 Laravel 框架中掌握那么几个高级的部分，剩下的也就很好理解了。

Laravel 框架给人的第一印象就是新颖，但当你深入去挖掘代码的内涵时，发现它所用的设计方法和理念又都是已经存在的，甚至很多都是我们了解的。但了解和应用还存在一定的差距，融会贯通、出神入化指出了对知识掌握程度的会、通、神、化四个境界，会境界和化境界差距就在这里。Laravel 框架无疑是将 PHP 的新特性及面向对象设计模式用到了一定高度，所以我们觉得它新颖，归根到底还是因为我们的思想陈旧。就如同编程语言，不同的语言间编程效率差距很大，当你没有学一门更好的编程语言之前，你一直以为自己所用的是最好的。本章将剖析 Laravel 框架中的设计模式，通过对框架中源码的分析，提高我们对设计模式的认识。

6.1 服务容器

在 Laravel 框架中，服务容器是整个系统功能调度配置的核心，将其称为 Laravel 框架的“心脏”也不为过，因为它提供了整个框架运行过程中需要的一系列服务。这里说服务容器可能读者很难从字面理解其功能含义，容器通俗的理解就是装东西的物体，常见的一个变量、一个对象等都可以看成一个容器，而服务容器就是提供服务的载体。在这里，我们可以将服务理解为系统运行中需要的东西，如对象、文件路径、系统配置等，服务容器就是这些东西的载体，在程序运行过程中动态地为系统提供这些服务，也可以看做是提供这些资源。

6.1.1 依赖与耦合

上面对服务容器做了简单介绍，服务容器可以提供很多内容，而在这些功能中，解决依赖实现解耦才是应用服务容器最关键的原因。这里不得不介绍 IoC（Inversion Of Control，控制反转）容器的概念，其实服务容器就相当于一个 IoC 容器，而说到 IoC 容器就要从控制反转模式说起。控制反转模式主要是用来解决系统组件间相互依赖关系的一种模式。于是，

我们得到应用这些技术的根源，解决依赖。什么是依赖？下面给出依赖的实例。

```
<?php
// 设计公共接口
interface Visit
{
    public function go();
}
// 实现不同交通工具类
class Leg implements Visit
{
    public function go()
    {
        echo "walt to Tibet!!!";
    }
}
class Car implements Visit
{
    public function go()
    {
        echo "drive car to Tibet!!!";
    }
}
class Train implements Visit
{
    public function go()
    {
        echo "go to Tibet by train!!!";
    }
}
// 设计旅游者类，该类在实现游西藏的功能时要依赖交通工具实例
class Traveller
{
    protected $trafficTool;
    public function __construct()
    {
        // 依赖产生
        $this->trafficTool = new Leg();
    }
    public function visitTibet()
    {
        $this->trafficTool->go();
    }
}
$tra = new Traveller();
```

```
$tra->visitTibet();
```

这里要实现的功能是旅游者游西藏，而游西藏可能有很多种方法，如走路、坐火车或开车，不同的方法需要不同的工具，如腿、火车和小汽车。在交通基本靠走的时代，游西藏用的交通工具就是腿，在代码“`$this->trafficTool = new Leg();`”中实例化了腿，于是上述代码两个组件间就产生了依赖。在程序中依赖可以理解为一个对象实现某个功能需要其他对象相关功能的支持。当用 `new` 关键字在一个组件内部实例化一个对象时就解决了一个依赖，但同时也引入了另一个严重的问题——耦合。在简单情况下可能看不出耦合有多大问题，但是如果需求改变，如需要实例化的交通工具是火车、汽车甚至是在设计中还不太清楚的工具时，就需要经常改变实例化的对象，如果又有很多地方用到了这段代码，而这个程序又不是你自己写的，这时你面对的将是噩梦。所以，这里我们不应该在“旅游者”类的内部固化“交通工具”的初始化行为，而是转由外部负责，在系统运行期间，将这种依赖关系通过动态注入的方式实现，这就是 IOC 模式的设计思想。

6.1.2 工厂模式

在上面实例中我们知道，交通工具的实例化过程是经常需要改变的，所以我们将这部分提取到外部来管理，这也就体现了面向对象设计的一个原则，即找出程序中会变化的方面，然后将其和固定不变的方面相分离（Head First 设计模式）。一种简单的实现方案是利用工厂模式，这里我们只用简单工厂模式实现，当然也可以用工厂方法模式。实例如下：

```
// 设计简单工厂，对于不同的输入，实例化不同的交通工具
```

```
class TrafficToolFactory
{
    public function createTrafficTool($name)
    {
        switch ($name) {
            case 'Leg':
                return new Leg();
                break;
            case 'Car':
                return new Car();
                break;
            case 'Train':
                return new Train();
                break;
            default:
                exit("set trafficTool error!!!");
                break;
        }
    }
}
```



```

class Traveller
{
    protected $trafficTool;
    public function __construct($trafficTool)
    {
        // 通过工厂产生依赖的交通工具实例
        $factory = new TrafficToolFactory();
        $this->trafficTool = $factory->createTrafficTool($trafficTool);
    }
    public function visitTibet()
    {
        $this->trafficTool->go();
    }
}

$tra = new Traveller('Train');
$tra->visitTibet();

```

这里我们添加了“交通工具”工厂，在“旅游者”实例化的过程中指定需要的交通工具，则工厂生产相应的交通工具实例。在一些简单的情况下，简单工厂模式可以解决这个问题。我们看到虽然“旅游者”和“交通工具”之间的依赖关系没有了，但是却变成了“旅游者”和“交通工具工厂”之间的依赖。当需求增加时，我们需要修改简单工厂模式，如果依赖增多，工厂将十分庞大，依然不易于维护。

6.1.3 IoC 模式

这时，我们今天的主角登场了。IoC（Inversion of Control）模式又称依赖注入（Dependency Injection）模式。控制反转是将组件间的依赖关系从程序内部提到外部容器来管理，而依赖注入是指组件的依赖通过外部以参数或其他形式注入，两种说法其实本质上是一个意思。下面是一个简单的依赖注入的实例：

```

class Traveller
{
    protected $trafficTool;
    public function __construct(Visit $trafficTool)
    {
        $this->trafficTool = $trafficTool;
    }
    public function visitTibet()
    {
        $this->trafficTool->go();
    }
}

// 生成依赖的交通工具实例

```

```

$trafficTool = new Leg();
// 依赖注入的方式解决依赖问题
$tra = new Traveller($trafficTool);
$tra->visitTibet();

```

上述实例就是一个依赖注入的过程，Traveller 类的构造函数依赖一个外部的具有 visit 接口的实例，而在实例化 Traveller 时，我们传递一个 \$trafficTool 实例，即通过依赖注入的方式解决依赖问题。这里要注意一点，依赖注入需要通过接口来限制，而不能随意开放，这也体现了设计模式的另一个原则——针对接口编程，而不是针对实现编程。

这里我们是通过手动的方式注入依赖，而通过 IoC 容器可以实现自动依赖注入。下面对 Laravel 框架中的设计方法进行了简化，实现 IoC 容器完成依赖注入，代码如下：

```

// 设计容器类，容器类装实例或提供实例的回调函数
class Container
{
    // 用于装提供实例的回调函数，真正的容器还会装实例等其他内容
    // 从而实现单例等高级功能
    protected $bindings = [];
    // 绑定接口和生成相应实例的回调函数
    public function bind($abstract, $concrete = null, $shared = false)
    {
        if ( ! $concrete instanceof Closure){
            // 如果提供的参数不是回调函数，则产生默认的回调函数
            $concrete = $this->getClosure($abstract, $concrete);
        }
        $this->bindings[$abstract] = compact('concrete', 'shared');
    }
    // 默认生成实例的回调函数
    protected function getClosure($abstract, $concrete)
    {
        // 生成实例的回调函数，$c 一般为 IoC 容器对象，在调用回调生成实例时提供
        // 即 build 函数中的 $concrete($this)
        return function($c) use ($abstract, $concrete)
        {
            $method = ($abstract == $concrete) ? 'build' : 'make';
            // 调用的是容器的 build 或 make 方法生成实例
            return $c->$method($concrete);
        };
    }
    // 生成实例对象，首先解决接口和要实例化类之间的依赖关系
    public function make($abstract)
    {
        $concrete = $this->getConcrete($abstract);
    }
}

```

```

        if ($this->isBuildable($concrete, $abstract)) {
            $object = $this->build($concrete);
        }
        else {
            $object = $this->make($concrete);
        }
        return $object;
    }

    protected function isBuildable($concrete, $abstract)
    {
        return $concrete === $abstract || $concrete instanceof Closure;
    }

    // 获取绑定的回调函数
    protected function getConcrete($abstract)
    {
        if ( ! isset($this->bindings[$abstract]))
        {
            return $abstract;
        }
        return $this->bindings[$abstract]['concrete'];
    }

    // 实例化对象
    public function build($concrete)
    {
        if ($concrete instanceof Closure){
            return $concrete($this);
        }
        $reflector = new ReflectionClass($concrete);
        if ( ! $reflector->isInstantiable()){
            echo $message = "Target [$concrete] is not instantiable.";
        }
        $constructor = $reflector->getConstructor();
        if (is_null($constructor)) {
            return new $concrete;
        }
        $dependencies = $constructor->getParameters();
        $instances = $this->getDependencies($dependencies);
        return $reflector->newInstanceArgs($instances);
    }

    // 解决通过反射机制实例化对象时的依赖
    protected function getDependencies($parameters)
    {
        $dependencies = [];
        foreach ($parameters as $parameter)
    
```



```

        {
            $dependency = $parameter->getClass();
            if (is_null($dependency)){
                $dependencies[] = NULL;
            }
            else{
                $dependencies[] = $this->resolveClass($parameter);
            }
        }
        return (array) $dependencies;
    }

    protected function resolveClass(ReflectionParameter $parameter)
    {
        return $this->make($parameter->getClass()->name);
    }
}

class Traveller
{
    protected $trafficTool;
    public function __construct(Visit $trafficTool)
    {
        $this->trafficTool = $trafficTool;
    }
    public function visitTibet()
    {
        $this->trafficTool->go();
    }
}

// 实例化 IoC 容器
$app = new Container();
// 完成容器的填充
$app->bind("Visit","Train");
$app->bind("traveller","Traveller");
// 通过容器实现依赖注入，完成类的实例化
$tra = $app->make("traveller");
$tra->visitTibet();

```

通过上例可以看到，仅仅百十行代码，就可以实现 IoC 容器最核心的功能，解决依赖注入的根本问题。在实现过程中，没有用 new 关键字来实例化对象，不需要人来关注组件间的依赖关系，只要在容器填充过程中理顺接口与实现类之间的关系及实现类与依赖接口之间的关系，就可以流水线式地完成实现类的实例化过程。这里类的实例化是通过反射机制完成的，对此不清楚的读者可以查看 PHP 的重要性质这一章关于反射机制的介绍，当然也可以在容器中直接填充实例化对象的回调函数，对于那些特殊的类可以设计特定的回调函数。

在 Laravel 框架的官方文档中，将其称为服务容器，核心功能是 IoC 容器用以解决依赖注入，而对服务容器的填充部分称为服务提供者，所以对于 Laravel 框架来说这种叫法更贴切，因为框架中的 Container 类并不仅仅完成了 IoC 容器的功能，还在程序运行过程中提供各种相应的服务，包括对象、生成对象的回调函数、配置等。在 Laravel 框架中容器类的设计要比这里复杂一些，因为需要实现更加复杂的功能，包括实现单例模式、生成实例消息处理等。

6.1.4 源码解析

在 Laravel 框架中，服务容器是通过 Illuminate\Container\Container 类来实现的，其实现原理与上述实例相同，这里给出该容器类的工作示意图，如图 6.1 所示。需要说明的是，服务绑定有时也称为服务注册，在全文中两者意义相同，只是对于不同上下文环境某种说法更加贴切而已。

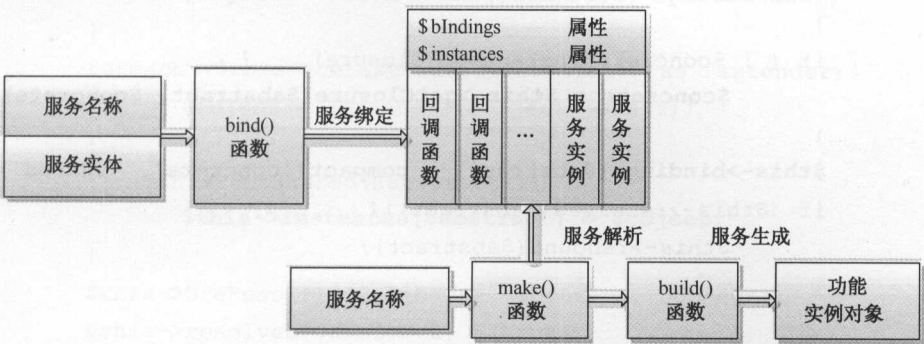


图 6.1 服务容器工作示意图

对于程序设计来说，源码是最好的老师，一切的概念通过描述或者加工后，都会存在意义上的偏差，只有通过了解源码才能真正领会其中的含义。这里给出 Laravel 框架中关于服务容器类实现的部分源码，通过与上面实例的对照，并结合 Laravel 框架容器类的示意图，进一步理解实现的方法和思想，加深对 IoC 等概念的理解。具体代码如下：

文件 Illuminate\Container\Container.php

```
// 容器绑定数组
protected $bindings = [];
// 容器共享实例数组
protected $instances = [];
```

服务容器类中定义了两个用于管理服务的属性，分别是 \$bindings 和 \$instances，其中 \$bindings 用于存储提供服务的回调函数，而 \$instances 用于存储程序中共享的实例，也可以称为单例。

文件 Illuminate\Container\Container.php

```
// 注册一个单例绑定到容器中
```

```

public function singleton($abstract, $concrete = null)
{
    $this->bind($abstract, $concrete, true);
}
// 注册一个绑定到容器中 , shared=true 表示是一个共享的绑定, 即 singleton
public function bind($abstract, $concrete = null, $shared = false)    如果
是
{
    if (is_array($abstract)){
        list($abstract, $alias) = $this->extractAlias($abstract);
        $this->alias($abstract, $alias);
    }
    $this->dropStaleInstances($abstract);
    if (is_null($concrete)){
        $concrete = $abstract;
    }
    if ( ! $concrete instanceof Closure)    {
        $concrete = $this->getClosure($abstract, $concrete);
    }
    $this->bindings[$abstract] = compact('concrete', 'shared');
    if ($this->resolved($abstract)){
        $this->rebound($abstract);
    }
}
// 得到一个闭包用于绑定一个类
protected function getClosure($abstract, $concrete)
{
    return function($c, $parameters = []) use ($abstract, $concrete)
    {
        $method = ($abstract == $concrete) ? 'build' : 'make';
        return $c->$method($concrete, $parameters);
    };
}

```

这几个函数实现了 Laravel 框架中服务容器的服务绑定功能, 主要是由 bind() 函数实现的。singleton() 函数实现的是单例绑定, 即程序中如果没有服务名称对应的实例对象, 则通过服务容器实例化一个后并进行记录, 如果在后续程序中还需要同名的服务时则返回先前创建的服务实例对象。该函数相当于 bind() 函数的一个特例, 即参数 \$shared 值为 true 的情况。对于 bind() 函数实现的服务绑定功能, 在忽略 \$shared 参数的情况下, 即不讨论单例还是普通的服务, 可以分为两种情况, 如果参数 \$concrete 为一个回调函数, 则直接将回调函数与服务名称 \$abstract 进行绑定; 如果参数 \$concrete 为一个名称, 则首先需要通过 getClosure() 函数创建服务回调函数, 然后将该回调函数与服务名称绑定, 总之需要实现一个可以生成相

应服务实例对象的回调函数与服务名称进行绑定。接下来介绍服务解析的实现，代码如下：

文件 Illuminate\Container\Container.php

// 通过容器实例化一个给定的类

```
public function make($abstract, $parameters = [])
```

```
{
    $abstract = $this->getAlias($abstract);
    if (isset($this->instances[$abstract])){
        return $this->instances[$abstract];
    }
    $concrete = $this->getConcrete($abstract);
    if ($this->isBuildable($concrete, $abstract)) {
        $object = $this->build($concrete, $parameters);
    }else{
        $object = $this->make($concrete, $parameters);
    }
    foreach ($this->getExtenders($abstract) as $extender){
        $object = $extender($object, $this);
    }
    if ($this->isShared($abstract)){
        $this->instances[$abstract] = $object;
    }
    $this->fireResolvingCallbacks($abstract, $object);
    $this->resolved[$abstract] = true;
    return $object;
}
```

// 获取抽象类名称的别名

```
protected function getAlias($abstract)
```

```
{
    return isset($this->aliases[$abstract]) ? $this->aliases[$abstract] :
```

```
$abstract;
```

```
}
// 判断给定的服务实体是否可以创建
```

```
protected function isBuildable($concrete, $abstract)
```

```
{
    return $concrete === $abstract || $concrete instanceof Closure;
```

```
}
// 根据抽象类名称获取实体类
```

```
protected function getConcrete($abstract)
```

```
{
    if ( ! is_null($concrete = $this->getContextualConcrete($abstract))){
        return $concrete;
    }
}
```

```

        if ( ! isset($this->bindings[$abstract])){
            if ($this->missingLeadingSlash($abstract) &&
                isset($this->bindings['\\'.$abstract])){
                $abstract = '\\'.$abstract;
            }
            return $abstract;
        }
        return $this->bindings[$abstract]['concrete'];
    }
}

```

服务解析过程略微复杂一点，可以将其分为两个步骤来完成，一个是完成对应服务的查找，另一个是完成服务的实现，一般是指完成实例化对象的创建。这两个步骤分别由 `make()` 和 `build()` 函数完成。

首先介绍服务查找过程，即由 `make()` 函数实现的功能。该函数需要提供两个参数，分别是 `$abstract` 和 `$parameters`，`$abstract` 可以看做是服务名称，而 `$parameters` 是创建实例化对象需要的参数，即一个类实例化时的依赖。对于服务的查找是根据服务名称 `$abstract` 来进行的，首先通过 `getAlias()` 函数来查找服务名称是否有别名，对于服务别名的管理是通过服务容器类中的 `$aliases` 数组属性实现的，而内容基本是通过 `Illuminate\Foundation\Application` 类中的 `registerCoreContainerAliases()` 函数注册的，如一个简单的实例，`Illuminate\Contracts\Container\Container` 抽象类的别名为“app”，如果查找到了别名，将查找该别名对应的服务，如果该抽象类没有别名，则继续进行查找。然后在服务容器的共享实例数组（`$instances` 属性）中查找服务名称的实例，如果查找到则说明该服务名称对应为单例，直接返回先前实例化的对象，否则继续查询。接下来，会通过 `getConcrete()` 获取服务名称的实体，在服务绑定时，一个服务名称一般绑定一个回调函数用于生成实例对象，而这个回调函数就相当于服务名称的实体。这个实体的查找就是通过容器中的 `$bindings` 数组属性实现的，如果查找到则返回实体，否则修改服务名称的形式继续下一次的查找。然后，会通过 `isBuildable()` 函数判断服务实体能否创建实例化对象，如果可以则转到下一个步骤，否则继续通过 `make()` 函数来查找。在完成实例对象的创建后，通过 `isShared()` 判断该服务是否为单例，如果是需要在共享实例对象数组（`$instances`）中记录。下面介绍实例化对象的创建步骤，代码如下：

文件 `Illuminate\Container\Container.php`

// 根据给定服务具体名称实例化一个具体类对象

```

public function build($concrete, $parameters = [])
{
    if ($concrete instanceof Closure){
        return $concrete($this, $parameters);
    }
    $reflector = new ReflectionClass($concrete);
    if ( ! $reflector->isInstantiable()){

```

```

        $message = "Target [$concrete] is not instantiable.";
        throw new BindingResolutionException($message);
    }
    $this->buildStack[] = $concrete;
    $constructor = $reflector->getConstructor();
    if (is_null($constructor)){
        array_pop($this->buildStack);
        return new $concrete;
    }
    $dependencies = $constructor->getParameters();
    $parameters = $this->keyParametersByArgument(
        $dependencies, $parameters
    );
    $instances = $this->getDependencies(
        $dependencies, $parameters
    );
    array_pop($this->buildStack);
    return $reflector->newInstanceArgs($instances);
}
// 通过反射机制解决所有的参数依赖
protected function getDependencies($parameters, array $primitives = [])
{
    $dependencies = [];
    foreach ($parameters as $parameter){
        $dependency = $parameter->getClass();
        if (array_key_exists($parameter->name, $primitives)) {
            $dependencies[] = $primitives[$parameter->name];
        }elseif (is_null($dependency)){
            $dependencies[] = $this->resolveNonClass
($parameter);
        }else{
            $dependencies[] = $this->resolveClass($parameter);
        }
    }
    return (array) $dependencies;
}
// 解决一个没有类提示的依赖
protected function resolveNonClass(ReflectionParameter $parameter)
{
    if ($parameter->isDefaultValueAvailable()){
        return $parameter->getDefaultValue();
    }
    // 省略异常处理部分内容
}

```



```
// 通过容器解决一个类的依赖
protected function resolveClass(ReflectionParameter $parameter)
{
    try{
        return $this->make($parameter->getClass()->name);
    }
    // 省略异常处理部分内容
}
```

在通过 `make()` 函数查找到服务实体后，会将其传递给 `build()` 函数用于对象的创建，如果服务实体就是一个闭包函数，则直接调用该闭包函数完成服务实例化对象的创建，如果服务实体只是一个具体类的类名，则需要通过反射机制来完成实例化对象的创建。通过反射机制完成对象实例化的过程，首先是根据将要实例化的类名称获取反射类 (`ReflectionClass`) 实例，然后获取该类在实例化过程中的依赖，即构造函数需要的参数，在 `build()` 函数中，通过 `getDependencies()` 函数来实现依赖的生成，如果在服务解析时提供了相应的参数，即通过 `$parameters` 参数提供，则直接使用提供的参数，如果没有提供，则通过服务容器中的 `resolveNonClass()` 函数来获取默认参数，或者通过 `resolveClass()` 函数来创建，而创建的方式也是通过服务容器，所以服务容器解决依赖注入的问题就是通过这部分代码实现的。在解决了依赖的问题后，可以直接通过反射机制完成服务实例对象的创建。

通过在 Laravel 框架源码的基础上分析服务容器的实现过程，读者应该对服务容器的概念、IOC 模式及依赖注入等概念有了进一步的了解，在后续章节中，还会在实际应用中多次遇到这些概念，掌握这部分内容对于后面了解 Laravel 框架的工作过程将十分有益。

6.2 请求处理管道简介

上面已经解决了 Laravel 框架中一个关键技术，就是利用服务容器和服务提供者解决依赖注入及资源获取的功能，有了它就可以随时获取需要的服务，实现想要的功能。但对于服务器来说，真正要实现的功能是处理输入的请求，并将生成的响应输出给客户端，而在处理请求的过程中需要经历很多处理步骤，这些步骤需要做到松耦合，可以随时在这些步骤中间添加新的处理功能而使改动尽可能小，通过源码的注解，设计者将其比喻成“洋葱”，像它一样分很多层，每一层具有一定的功能，可以随时添加或修改这些层，而官方文档中将这层称为中间件，通过这些中间件使得程序的可扩展性大大增强。这里，其实使用的是一种装饰者模式，只是 PHP 特有的编程方式使得其形式发生变化，下面我们就逐步揭开它的面纱。

6.2.1 装饰者模式

装饰者模式是在开放—关闭原则下实现动态添加或减少功能的一种方式。以 Laravel 框架为例，在解析请求生成响应之前或之后需要经过中间件的处理，主要包括验证维护模式、Cookie 加密、开启会话、CSRF 保护等，而这些处理有些是在生成响应之前，有些是在生成响应之后，在实际开发过程中有可能还需要添加新的处理功能，如果不修改原有类的基础上动态地添加或减少处理功能将使框架可扩展性大大增强，而这种需求正好可以被装饰者模式解决。下面简单给出一个装饰者模式的实例，因为本书不是主要介绍设计模式的，所以无法理解的读者可以查阅关于设计模式方面的书籍。

```
<?php
interface Decorator
{
    public function display();
}

class XiaoFang implements Decorator
{
    private $name;
    public function __construct($name)
    {
        $this->name = $name;
    }
    public function display()
    {
        echo "我是 ".$this->name." 我出门了!!! ".<br>';
    }
}

class Finery implements Decorator
{
    private $component;
    public function __construct(Decorator $component)
    {
        $this->component = $component;
    }
    public function display()
    {
        $this->component->display();
    }
}

class Shoes extends Finery
{
    public function display()
    {
```

```

        echo " 穿上鞋子 " . '<br>';
        parent::display();
    }
}

class Skirt extends Finery
{
    public function display()
    {
        echo " 穿上裙子 " . '<br>';
        parent::display();
    }
}

class Fire extends Finery
{
    public function display()
    {
        echo ' 出门前先整理头发 ' . '<br>';
        parent::display();
        echo ' 出门后再整理一下头发 ' . '<br>';
    }
}

$xiaofang = new XiaoFang('小芳');
$shoes = new Shoes($xiaofang);
$skirt = new Skirt($shoes);
$fire = new Fire($skirt);
$fire->display();
输出:
出门前先整理头发
穿上裙子
穿上鞋子
我是小芳, 我出门了
出门后再整理一下头发

```

我们假设小芳接到一个电话算是请求, 而小芳出门是对请求的响应, 那么在小芳出门前后要对自己进行打扮, 对应于 Laravel 框架中, 这些打扮的步骤就相当于中间件的功能, 而小芳出门是对请求的真正响应。在小芳打扮的过程中, 可以随时增加新的打扮类, 只要该类继承 Finery 类 (装饰类) 并调用父类的同名方法, 就可以在实现时重新组织打扮过程, 实现打扮步骤的增加或减少, 例如加一件衣服、化个妆等。这就是装饰者模式的应用场景。

6.2.2 请求处理管道

增加或减少功能需要重新组织相应过程, 即实例化的顺序, 因为这里实例化过程是手动实现的。手动, 我们想到了什么, 对, 就是服务容器, 在上一节已经讲了这个解决依赖注

人的自动化设备，而 Laravel 框架就是通过服务容器进行自动实例化的，实例间的功能调用也是通过闭包函数完成的，这里为了将问题简单化，我们通过静态函数来避免实例化的过程，只仿真通过闭包函数完成装饰者模式，实现请求的处理管道。在 Laravel 框架中，针对请求的处理过程一共使用三次处理管道，这部分我们将在第 7 章中介绍。下面我们先看一段管道代码：

```
<?php
interface Middleware
{
    public static function handle(Closure $next);
}
class VerifyCsrfToken implements Middleware
{
    public static function handle(Closure $next)
    {
        echo " 验证 Csrf-Token".'<br>';
        $next();
    }
}
class ShareErrorsFromSession implements Middleware
{
    public static function handle(Closure $next)
    {
        echo " 如果 session 中有 'errors' 变量，则共享它 ".'<br>';
        $next();
    }
}
class StartSession implements Middleware
{
    public static function handle(Closure $next)
    {
        echo " 开启 session，获取数据 ".'<br>';
        $next();
        echo " 保存数据，关闭 session".'<br>';
    }
}
class AddQueuedCookiesToResponse implements Middleware
{
    public static function handle(Closure $next)
    {
        $next();
        echo " 添加下一次请求需要的 cookie".'<br>';
    }
}
```

```

class EncryptCookies implements Middleware
{
    public static function handle(Closure $next)
    {
        echo " 对输入请求的 cookie 进行解密 " . '<br>';
        $next();
        echo " 对输出响应的 cookie 进行加密 " . '<br>';
    }
}

class CheckForMaintenanceMode implements Middleware
{
    public static function handle(Closure $next)
    {
        echo " 确定当前程序是否处于维护状态 " . '<br>';
        $next();
    }
}

function getSlice()
{
    return function($stack, $pipe)
    {
        return function() use ($stack, $pipe)
        {
            return $pipe::handle($stack);
        };
    };
}

function then()
{
    $pipes = [
        "CheckForMaintenanceMode",
        "EncryptCookies",
        "AddQueuedCookiesToResponse",
        "StartSession",
        "ShareErrorsFromSession",
        "VerifyCsrfToken"];
    $firstSlice = function(){
        echo " 请求向路由器传递, 返回响应 " . '<br>';
    };
    $pipes = array_reverse($pipes);
    call_user_func(
        array_reduce($pipes, getSlice(), $firstSlice)
    );
}

```

```
then();
```

输出:

确定当前程序是否处于维护状态

对输入请求的 cookie 进行解密

开启 session, 获取数据

如果 session 中有 'errors' 变量, 则共享它

验证 Csrf-Token

请求向路由器传递, 返回响应

保存数据, 关闭 session

添加下一次请求需要的 cookie

对输出响应的 cookie 进行加密

上面的输出内容是 Laravel 框架对请求处理的部分流程, 这里面大部分与上一节中装饰者模式形式相似, 但通过回调函数生成整个处理流程的过程还是比较难以理解。这里给一个简单的实例用于理解, 代码如下:

```
<?php
interface Step {
    public static function go(Closure $next);
}

class FirstStep implements Step {
    public static function go(Closure $next)
    {
        echo " 开启 session, 获取数据 " . '<br>';
        $next();
        echo " 保存数据, 关闭 session " . '<br>';
    }
}

function goFun($step, $className)
{
    return function() use($step, $className)
    {
        return $className::go($step);
    };
}

function then()
{
    $steps = ["FirstStep"];
    $prepare = function(){echo " 请求向路由器传递, 返回响应 " . '<br>'}};
    $go = array_reduce($steps, "goFun", $prepare);
    $go();
}

then();
输出:
```


开启 session, 获取数据
请求向路由器传递, 返回响应
保存数据, 关闭 session

这里我们将处理功能减少为一步, 整个程序比较难理解的是 `array_reduce($steps, "goFun", $prepare)` 函数和 `goFun($step, $className)` 函数, 其中 `array_reduce()` 函数在参数手册中介绍是用回调函数迭代地将数组内容进行处理, 共有三个参数, 前两个参数是必须赋值的, 第一个是要处理的数组, 第二个是处理函数名称或回调函数, 第三个参数为可选参数, 为初始化参数, 将被当做数组中第一个值来处理, 如果数组为空则作为返回值。这里我们给第三个参数传递一个回调函数, 该函数用于将请求向路由器继续传递, 返回响应, 而第一个参数为一个数组, 该数组记录了外层功能的类名, `goFun()` 函数作为处理数组的回调函数。`array_reduce()` 最终返回的是一个回调函数, 即 `$go`, 代码如下:

```
$go = function()  
{  
    return $FirstStep::go(function(){echo " 请求向路由器传递, 返回响应  
".'<br>';});  
};
```

在前面的例子中, 通过 `call_user_func()` 函数执行这个回调函数, 其实就相当于 `$go()`。这里我们可以清晰地理解请求处理通道是如何设计出来的。笔者第一次接触这部分也是很难理解, 简化一下就显而易见了, 所以复杂的东西之所以复杂是我们没有把它简化。

6.2.3 部分源码

在 Laravel 框架中, 请求处理管道主要是通过 `Illuminate\Pipeline\Pipeline` 类实现的, 其处理流程示意图如图 6.2 所示。

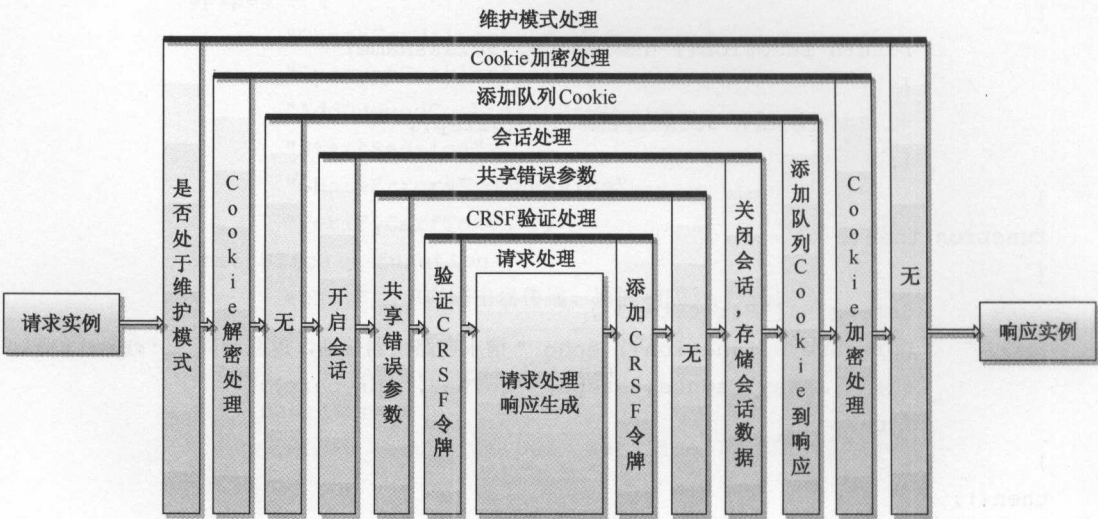


图 6.2 管道请求处理流程示意图

这里依然给出这部分内容在 Laravel 框架中对应的源码，一方面有个对照和比较，另一方面熟悉一下 Laravel 框架中的代码：

文件 `laravel\public\index.php`

```
<?php
$app = require_once __DIR__.'../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
$response = $kernel->handle(
    $request = Illuminate\Http\Request::capture()
);
```

该文件中 `$app` 即为第一节所说的服务容器，利用服务容器生成 `Kernel` 类的实例，即 `$app->make(Illuminate\Contracts\Http\Kernel::class)`。通过调用“`$kernel->handle($request = Illuminate\Http\Request::capture());`”来处理请求，返回请求的响应。

文件 `Illuminate\Foundation\Http\Kernel.php`

```
// 处理输入的 HTTP 请求
public function handle($request)
{
    try {
        $request->enableHttpMethodParameterOverride();
        $response = $this->sendRequestThroughRouter($request);
    } catch (Exception $e) {
        // 省略部分错误处理代码
        $this->app['events']->fire('kernel.handled', [$request, $response]);
        return $response;
    }
    // 通过中间件和路由处理请求
    protected function sendRequestThroughRouter($request)
    {
        $this->app->instance('request', $request);
        Facade::clearResolvedInstance('request');
        $this->bootstrap();
        return (new Pipeline($this->app))
            ->send($request)
            ->through($this->app->shouldSkipMiddleware() ? [] :

        $this->middleware)
            ->then($this->dispatchToRouter());
    }
}
```

文件 `Illuminate\Foundation\Application.php`

```
// 判断本程序的中间件是否被使能
```

```

public function shouldSkipMiddleware()
{
    return $this->bound('middleware.disable') &&
        $this->make('middleware.disable') === true;
}
// 获得路由分发器的回调函数
protected function dispatchToRouter()
{
    return function ($request) {
        $this->app->instance('request', $request);
        return $this->router->dispatch($request);
    };
}

```

上面是 Kernel 类，请求需要经过 handle() 函数进行处理并返回请求的响应，而 handle() 函数首先将请求实例传递给 sendRequestThroughRouter(\$request) 函数，该函数首先通过代码 “\$this->bootstrap();” 实现程序运行前的准备工作，包括环境检测、配置加载、日志管理、异常处理、外观注册、服务提供者注册、启动服务提供者七个步骤，这部分内容在以后还会详细介绍，这里只需要知道它完成了处理请求前的程序初始化工作。然后会新建一个 Pipeline 类，并依次调用 send()、through() 和 then() 函数，前两个函数比较简单，只是将请求实例 \$request 和中间件数组 \$this->middleware 传递给 Pipeline 类实例，在获取中间件的时候先要通过服务容器中的 shouldSkipMiddleware() 函数判断本程序是否使能中间件，如果需要中间件将添加该类中的中间件数组 (\$middleware 属性)，否则添加空数组。接着，then() 函数用来建立管道处理，其中参数 \$this->dispatchToRouter() 返回的是一个回调函数，这个回调函数可以对应上一小节实例中实现的“请求向路由器传递，返回响应”回调函数，而接下来的工作将交由 Pipeline 类实例来完成，我们继续看源码。

文件 Illuminate\Pipeline\Pipeline.php

```

// 实现管道处理和最终目标回调函数处理
public function then(Closure $destination)
{
    $firstSlice = $this->getInitialSlice($destination);
    $pipes = array_reverse($this->pipes);
    return call_user_func(
        array_reduce($pipes, $this->getSlice(), $firstSlice),
        $this->passable
    );
}
// 获取开始堆栈调用的最初片段
protected function getInitialSlice(Closure $destination)
{
    return function($passable) use ($destination)

```



```

        {
            return call_user_func($destination, $passable);
        };
    }
    // 获得一个闭包函数，函数中封装了中间件和路由处理的每一个片段
    protected function getSlice()
    {
        return function($stack, $pipe)
        {
            return function($passable) use ($stack, $pipe)
            {
                if ($pipe instanceof Closure){
                    return call_user_func($pipe, $passable,
$stack);
                }
                else{
                    return $this->container->make($pipe)
->{$this->method}($passable,
$stack);
                }
            };
        };
    }
}

```

这里依然从 `array_reduce($pipes, $this->getSlice(), $firstSlice)` 函数说起，很熟悉吧。其中，`$pipes` 是中间件类的名称数组，其定义在 `laravel/app/Http/Kernel.php` 文件中，即为 `$middleware` 属性，该数组参数是通过 `through()` 函数传递进来的，在前面的源码解析中提到过。而 `$firstSlice` 其实就是 `Kernel` 类中 `dispatchToRouter()` 函数返回的回调函数，只是这里给这个返回的回调函数传入了参数，这个参数此时是请求的实例，是通过 `send()` 函数传递进来的。接下来就是 `$this->getSlice()` 函数，该函数返回的是一个回调函数，用来处理中间件类，即生成一个处理管道的回调函数，这里用到了服务容器来实例化对象，即 `$this->container->make($name)`，而 `call_user_func_array([$this->container->make($name), $this->method], array_merge([$passable, $stack], $parameters))` 其实就是实现实例化中间类并调用该实例的 `handle` 函数来处理输出参数 `array_merge([$passable, $stack], $parameters)`。如果读者看代码还是不容易理解，可以稍微修改一下写出来，进行调试就能理解了。

本章从设计模式的角度剖析了 Laravel 框架使用的部分设计模式，设计模式简单地讲就是经过实践证明过的解决某一类编程问题的实现方法，这些方法简单而高效，使得成为一种约定俗成的编程经验，大的方面讲有框架的设计模式，如非常著名的 MVC，小的方面讲就是程序的设计模式，如本章中介绍的工厂模式、装饰者模式等，对设计模式有了深入了解后，对大部分编程问题都会有一套非常成熟的方法去解决，同时对于了解别人编写的代码也能从更大的粒度上认识，而不是从函数、代码级别去认识。

第7章

请求到响应的生命周期

前面已经讲了很多内容，但很多人看到这里可能还是一头雾水，因为前面都是讲 Laravel 框架的某个部分，虽然把有些部分讲得很细，但读者可能依然不知道 Laravel 框架到底是怎么工作的，仿佛是盲人摸象，只是摸了一小部分，在思维中产生的概念一定是不全面的。可能有人会问为什么不一开始就讲请求到响应的生命周期，这样一下子就能全面了解了。其实，如果基础的东西没有一个概念，讲整体依然不好理解，总之学习一项新技术总是要经历迷雾的阶段，这个阶段没有什么好方法，死记硬背记住一些概念就好了，不要急，很快就要清晰了。

当读者了解了 Laravel 框架的整个运行流程，清楚请求到响应的整个生命周期中每一步是用来做什么的、实现什么目标的，那么读者将对驾驭 Laravel 框架更有信心，对改造和排错更能胸有成足、了然于心。

7.1 程序启动准备

要了解 Laravel 框架的整个运行流程，那么首先要看的就是入口文件写了什么，再一步步向下探索整个脉络，就像读 C 语言程序首先要读 main 函数一样。在 Laravel 框架中，所有的请求入口文件是 public 目录下的 index.php 文件。当第一次看到入口文件时，可能会被它的简洁所震撼，这么简单的几行就清晰地展示了请求到响应的整个生命周期，下面先给出 index.php 文件的代码，然后将分别介绍。

文件 laravel/public/index.php

```
<?php
require __DIR__.'/../bootstrap/autoload.php';
$app = require_once __DIR__.'/../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
$response = $kernel->handle($request = Illuminate\Http\
Request::capture());
$response->send();
```

```
$kernel->terminate($request, $response);
```

程序的启动准备阶段是入口文件中的代码“require_once __DIR__.'../bootstrap/app.php’”部分，主要实现了服务容器的实例化和基本注册，包括服务容器本身注册、基础服务提供者注册、核心类别名注册和基本路径注册，在注册过程中，服务容器会在对应属性中记录注册的内容，以便在程序运行期间提供对应的服务。程序启动准备阶段具体过程如图 7.1 所示。

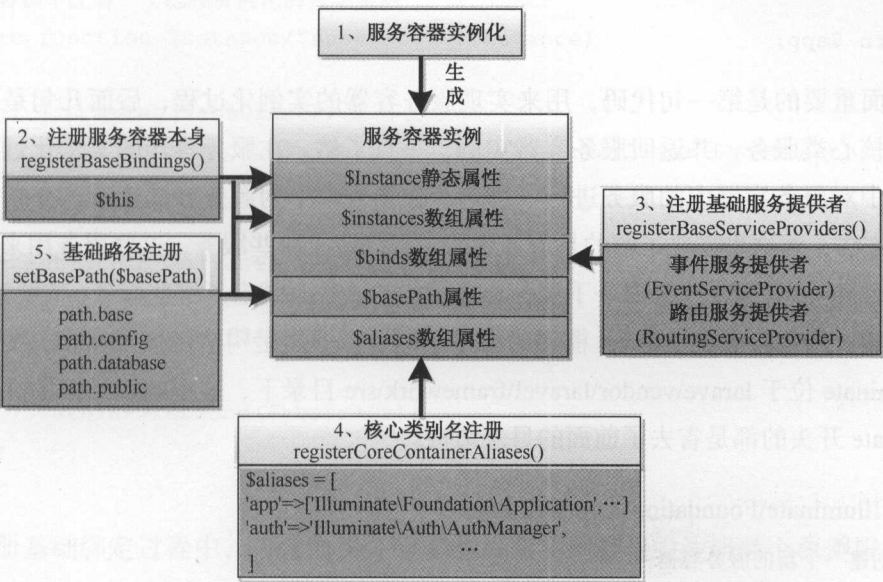


图 7.1 程序启动准备阶段流程

7.1.1 服务容器实例化

入口文件“iindex.php”的第一句包含 bootstrap 文件夹下的 autoload.php 文件，用来实现类的自动加载功能。文件的第二句是调用 bootstrap 文件夹下的 app.php 中的代码，主要用来实例化服务容器，并注册 Laravel 框架的核心类服务，为后面自动生成 \$kernel 核心类实例提供基础。这部分就是本小节将要介绍的请求到响应生命周期的准备阶段，接下来将详细介绍准备阶段都做了哪些工作。下面是准备阶段的源码：

文件 laravel\bootstrap\app.php

```
<?php
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'../..')
);
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);
```



```

);
$app->singleton(
    Illuminate\Contracts\Console\Kernel::class,
    App\Console\Kernel::class
);
$app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
);
return $app;

```

这里面重要的是第一句代码，用来实现服务容器的实例化过程，后面几句是向服务容器中绑定核心类服务，并返回服务容器实例，一目了然。在服务容器的实例化过程中，在构造函数中对服务容器中的服务进行了绑定。在第6章中的服务容器小节，介绍了服务容器的实现方法，而这里需要了解的是服务容器中都绑定了哪些服务，这些服务用来做什么，将在什么时候用到。那么，先看一下服务容器的构造函数都绑定了哪些服务。需要注意的是，服务容器的实例化参数是 Laravel 框架的根目录地址。下面是构造函数及其相关的源码，文件夹 `Illuminate` 位于 `larave\vendor\laravel\framework\src` 目录下，这里约定后面的文件路径中以 `Illuminate` 开头的都是省去了前面的目录结构。

文件 `Illuminate\Foundation\Application.php`

```

// 创建一个新的服务容器
public function __construct($basePath = null)
{
    $this->registerBaseBindings();
    $this->registerBaseServiceProviders();
    $this->registerCoreContainerAliases();
    if ($basePath) { $this->setBasePath($basePath); }
}

```

1. 注册基础绑定

服务容器第一句代码用来绑定基础服务，主要是绑定容器实例本身，使得其他的对象可以很容易得到服务容器实例，其中服务容器中设置了一个静态变量 `$instance`，该变量是在 `Container` 容器类中定义的，因为 `Application` 类继承了 `Container` 容器类，所以继承该静态变量，可以通过 `Container` 中的静态函数 `getInstance()` 直接获取服务容器实例。另外，为服务容器实例绑定了不同的服务别名，记录在 `$instances` 共享实例数组中，可以通过这些别名的任何一个找到服务容器实例。注册基础绑定的主要源码如下：

文件 `Illuminate\Foundation\Application.php`

```

// 向容器中注册基础绑定

```

```
protected function registerBaseBindings()
{
    static::setInstance($this);
    $this->instance('app', $this);
    $this->instance('Illuminate\Container\Container', $this);
}
```

文件 Illuminate\Container\Container.php

// 在容器中注册一个已经实例化的共享实例

```
public function instance($abstract, $instance)
{
    if (is_array($abstract)) {
        list($abstract, $alias) = $this->extractAlias($abstract);
        $this->alias($abstract, $alias);
    }
    unset($this->aliases[$abstract]);
    $bound = $this->bound($abstract);
    $this->instances[$abstract] = $instance;
    if ($bound) {
        $this->rebound($abstract);
    }
}
```

在注册基础绑定过程中，将会向服务容器的共享实例数组中注册两个单例服务，服务名称分别为“app”和“Illuminate\Container\Container”，对应的实例对象即为服务容器本身。

2. 注册基础服务提供者

接下来，将进行基础服务提供者的注册。服务提供者的注册是 Laravel 应用程序的启动和运行中最重要的行为之一，因为它为服务容器添加应用需要的各种服务，毕竟服务容器再强大，如果里面什么都没有装，那也是没有用的。在服务容器的构造函数中只注册了最基础的两个服务提供者，随着 Laravel 应用程序的运行还会有很多服务提供者被加载注册，这里我们先看看服务提供者是如何被注册的。

文件 Illuminate\Foundation\Application.php

// 注册基础服务提供者

```
protected function registerBaseServiceProviders()
{
    $this->register(new EventServiceProvider($this));
    $this->register(new RoutingServiceProvider($this));
}

// 在服务容器中注册一个服务提供者
public function register($provider, $options = [], $force = false)
{

```

```

        if ($registered = $this->getProvider($provider) && !$force) {
            return $registered;
        }
        if (is_string($provider)) {
            $provider = $this->resolveProviderClass($provider);
        }
        $provider->register();
        foreach ($options as $key => $value) {
            $this[$key] = $value;
        }
        $this->markAsRegistered($provider);
        if ($this->booted) {
            $this->bootProvider($provider);
        }
        return $provider;
    }
    // 如果服务提供者已经存在，则获取这个实例对象
    public function getProvider($provider)
    {
        $name = is_string($provider) ? $provider : get_class($provider);
        return Arr::first($this->serviceProviders, function ($key, $value)
        use ($name) {
            return $value instanceof $name;
        });
    }
    // 通过类名实例化一个服务提供者
    public function resolveProviderClass($provider)
    {
        return new $provider($this);
    }
    // 启动规定的服务提供者
    protected function bootProvider(ServiceProvider $provider)
    {
        if (method_exists($provider, 'boot')) {
            return $this->call([$provider, 'boot']);
        }
    }
}

```

我们来提炼一下基础服务提供者注册所必须的步骤，一是实例化服务提供者，这部分内容是通过 `resolveProviderClass()` 函数完成的；二是调用服务提供者的 `register()`，该函数用于向服务容器中注册服务；三是标识该服务提供者已经注册过了，这部分内容是通过 `markAsRegistered()` 函数实现的；四是如果应用程序已经启动过了，主要是程序中需要使用的服务提供者都已经注册完毕，则会调用服务提供者的 `boot()` 函数。每个服务提供者都继承

自 Illuminate\Support\ServiceProvider 类，该类有个 register() 虚函数，所以每个服务提供者必须实现这个函数，用来填充服务容器并提供服务。同时，应用程序会在适当时候统一调用服务提供者的 boot() 函数，但这个函数服务提供者可以不实现，因为在 Illuminate\Support\ServiceProvider 类中的魔术方法 __call() 提供了该函数的处理。所以，如果自己需要设计服务提供者来进行服务注册，则需要继承该类并实现这两个函数。

3. 注册核心类别名和应用的基础路径

由于 Laravel 框架的类是基于命名空间的，所以类名都比较长，为此在服务容器中为一些常用的类注册了别名，在后面程序中会通过别名来代替这个类名。基础路径是指应用程序关键目录的路径，服务容器中也注册这些信息。

文件 Illuminate\Foundation\Application.php

```
// 在容器中注册核心类的别名
public function registerCoreContainerAliases()
{
    $aliases = [
        'app' => ['Illuminate\Foundation\Application',
        'Illuminate\Contracts\Container\Container', 'Illuminate\Contracts\Foundation\
Application'],
        'auth' => 'Illuminate\Auth\AuthManager',
    // 这里省略了别名数组中部分内容
    ];
    foreach ($aliases as $key => $aliases) {
        foreach ((array) $aliases as $alias) {
            $this->alias($key, $alias);
        }
    }
}
// 注册应用的基础路径
public function setBasePath($basePath)
{
    $this->basePath = rtrim($basePath, '\\');
    $this->bindPathsInContainer();
    return $this;
}
// 在容器中绑定应用程序的基础路径
protected function bindPathsInContainer()
{
    $this->instance('path', $this->path());
    foreach (['base', 'config', 'database', 'lang', 'public',
'storage'] as $path) {
        $this->instance('path.'.$path, $this->{$path.'Path'}());
    }
}
```

可以看到，在 `registerCoreContainerAliases()` 的 `$aliases` 数组变量中定义了整个框架的核心服务别名，在服务解析过程中，需要根据实例化的类或接口名称查找服务别名，然后通过服务别名获取具体的服务。至此，应用程序的准备工作已经完成了，这里已经生成了服务容器，在服务容器中注册绑定了基础的服务提供者、服务别名和基础路径。

7.1.2 核心类（Kernel 类）实例化

服务容器实例化后，就可以通过服务容器来自动实例化对象了。于是，Kernel 类就通过服务容器自动化创建而成，即 `index.php` 文件中的 `"$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);"`。那么我们又在什么时候注册的服务呢？前面已经介绍过，在 `laravel\bootstrap\app.php` 文件中，实例化服务容器之后就注册了三个服务，其中就包括这个核心类接口。在注册服务时，服务名一般是接口。在 `Contracts` 命名空间下存储的都是接口，而提供的服务则是具体类、实例对象或返回实例对象的回调函数。

由于注册的服务只是具体类名，所以可以通过反射机制来实例化，并通过反射机制自动解决构造函数中的依赖关系。于是，通过服务容器实例化 `App\Http\Kernel` 类时，这个类只是定义了 `$middleware`（中间件）和 `$routeMiddleware`（路由中间件）两个数组属性，其中中间件是请求进入路由处理前的处理类，而路由中间件是请求进入路由处理后的处理类，所以这里可以添加新的中间件处理类，只要按照中间件的设计原则进行设计，并在中间件数组的正确位置添加类名，在处理请求的过程中就会调用新添加的中间件处理过程，这部分内容在 Laravel 框架的设计模式中具体介绍过。可以看到，Laravel 框架对于扩展就是这么简单，这也得益于框架设计的艺术性。

因为 `App\Http\Kernel` 类继承了 `Illuminate\Foundation\Http\Kernel` 类，所以实例化过程中会调用该类中的构造函数，下面是构造函数源码。

文件 `Illuminate\Foundation\Http\Kernel.php`

```
// 创建一个新的 HTTP 核心类实例
public function __construct(Application $app, Router $router)
{
    $this->app = $app;
    $this->router = $router;
    foreach ($this->routeMiddleware as $key => $middleware) {
        $router->middleware($key, $middleware);
    }
}
```

可以看到，Kernel 类的构造函数是存在依赖的，两个依赖分别是 `Illuminate\Contracts\`

Foundation\Application 和 Illuminate\Routing\Router 类型，这里依赖的类型要加上命名空间才能正确定义。其中，对于 Illuminate\Contracts\Foundation\Application 类名我们定义了别名，在上一小节的核心类别名中定义的，于是得到服务名为“app”，而名称为“app”的服务在上一小节注册基础绑定过程中已经注册了，即为服务容器的实例；对于 Illuminate\Routing\Router 类也注册了别名为“router”，而名为“router”的服务则是在注册基础服务提供者 RoutingServiceProvider 类的 register() 函数中注册的，即获取 Illuminate\Routing\Router 类的实例。在实例化路由类（Router）的时候还会存在依赖，依然通过服务容器自动生成。这里我们看到了服务提供者和服务容器是如何配合来完成依赖注入的整个过程。完成了服务容器和核心类（Kernel 类）的实例化之后，接下来该处理请求了，这才是我们的目标。

7.2 请求实例化

在 Laravel 框架中，完成准备工作后，将进行请求的实例化。什么是请求？其实就是客户端发送的一个请求报文，这个报文包括请求行、请求首部和请求实体。这部分内容在 HTTP 协议中进行了介绍，这些都可以看做参数和值的对应，在 Laravel 框架中我们将其分类并保存在 Illuminate\Http\Request 类的实例对象中，于是请求也就转换为一个实例对象，在处理请求的过程中，只需要处理这个实例对象就可以了。请求实例的创建是通过 Illuminate\Http\Request 类的 capture() 静态函数完成的，即 `$request = Illuminate\Http\Request::capture()`，这个函数的源码如下：

文件 Illuminate\Http\Request.php

```
// 通过服务器提供的变量创建一个 HTTP 请求实例
public static function capture()
{
    static::enableHttpMethodParameterOverride();
    return static::createFromBase(SymfonyRequest::createFromGlobals());
}
// 创建一个请求实例通过 Symfony 实例
public static function createFromBase(SymfonyRequest $request)
{
    if ($request instanceof static) {
        return $request;
    }
    $content = $request->content;
    $request = (new static)->duplicate(
        $request->query->all(), $request->request->all(), $request-
        >attributes->all(),
        $request->cookies->all(), $request->files->all(), $request->server-
```



```

>all()
    );
    $request->content = $content;
    $request->request = $request->getInputSource();
    return $request;
}

```

首先要说明一点，Laravel 框架的底层用了很多 Symfony 框架的功能模块，开源提倡“不必重复发明轮子”，Laravel 框架的设计者认为 Symfony 底层设计得足够好了，也就拿过来用，这也是模块化开发的优势，哪个模块好，拿过来就用，如果没有模块化开发理念、没有 PSR 规范、没有 composer 工具，这一切是无法实现的。其实 Laravel 框架也是借鉴了大量其他框架的设计理念后才开发出来的，包括 Symfony、ruby on rails 等。所以，在吸收了 Laravel 框架的设计理念后，读者也可以站在它的肩膀上创新了。

通过上面的代码可以看到，Laravel 框架的请求实例是在 Symfony 请求实例的基础上创建的。而 Symfony 框架的请求实例是通过 `createFromGlobals()` 静态函数实现的，接下来介绍 Symfony 框架对请求是如何进行实例化封装的，这也是请求实例化的重点。

文件 `vendor/symfony/http-foundation/Request.php`

// 通过 PHP 的全局变量创建一个新的请求实例

```

public static function createFromGlobals()
{
    $server = $_SERVER;
    if ('cli-server' === php_sapi_name()) {
        if (array_key_exists('HTTP_CONTENT_LENGTH', $_SERVER)) {
            $server['CONTENT_LENGTH'] = $_SERVER['HTTP_CONTENT_LENGTH'];
        }
        if (array_key_exists('HTTP_CONTENT_TYPE', $_SERVER)) {
            $server['CONTENT_TYPE'] = $_SERVER['HTTP_CONTENT_TYPE'];
        }
    }

    $request = self::createRequestFromFactory($_GET, $_POST, array(), $_COOKIE, $_FILES, $server);
    if (0 === strpos($request->headers->get('CONTENT_TYPE'), 'application/x-www-form-urlencoded') && in_array(strtoupper($request->server->get('REQUEST_METHOD', 'GET')), array('PUT', 'DELETE', 'PATCH'))) {
        parse_str($request->getContent(), $data);
        $request->request = new ParameterBag($data);
    }
    return $request;
}

```

在 Symfony 框架中，是通过 PHP 的全局数组作为参数来实例化请求的，其中包括 \$

GET、\$_POST、\$_COOKIE、\$_FILES 和 \$_SERVER，只是开始先对 \$_SERVER 中的参数进行了一下处理，因为 PHP 的一个 bug，当 PHP 的接口类型为 cli-server 时，会将 Content-Type 和 Content-Length 的值存储在 HTTP_CONTENT_TYPE 和 HTTP_CONTENT_LENGTH 两个字段中，这里需要对其进行修改，然后这些全局数组交给请求创建工厂。

// 请求创建工厂

```
private static function createRequestFromFactory(array $query = array(),
array $request = array(), array $attributes = array(), array $cookies =
array(), array $files = array(), array $server = array(), $content = null)
{
    if (self::$requestFactory) {
        $request = call_user_func(self::$requestFactory, $query,
        $request, $attributes, $cookies, $files, $server, $content);
        if (!$request instanceof self) {
            throw new \LogicException('The Request factory
            must return an instance of Symfony\Component\HttpFoundation\Request.');
```

如果自定义了请求工厂方法，则可以将自定义的工厂方法赋值给属性 \$requestFactory，否则将通过 new static 来完成请求的实例化。new static 的语法可以参看 PHP 的重要性质中的后期静态绑定部分。

// 构造函数

```
public function __construct(array $query = array(), array $request =
array(), array $attributes = array(), array $cookies = array(), array $files =
array(), array $server = array(), $content = null)
{
    $this->initialize($query, $request, $attributes, $cookies, $files,
    $server, $content);
}
// 初始化请求实例的参数
public function initialize(array $query = array(), array $request =
array(), array $attributes = array(), array $cookies = array(), array $files =
array(), array $server = array(), $content = null)
{
    $this->request = new ParameterBag($request);
    $this->query = new ParameterBag($query);
    $this->attributes = new ParameterBag($attributes);
    $this->cookies = new ParameterBag($cookies);
```

```

$this->files = new FileBag($files);
$this->server = new ServerBag($server);
$this->headers = new HeaderBag($this->server->getHeaders());
$this->content = $content;
// 省略部分参数初始化内容
}

```

请求相关信息的参数是通过 `ParameterBag`、`FileBag` 等类的实例来封装的，其中 `FileBag`、`ServerBage` 等类也继承了 `ParameterBag` 类，相应的参数存储方式是相同的，只是添加了更多的参数处理函数功能。

这里只是介绍了 Laravel 框架的请求实例化过程，在开发过程中还需要对请求的实例进行不同的操作，包括对请求参数的访问和存储等，这部分内容将在后续章节详细介绍。

7.3 处理请求

在完成了请求实例化后，将进入对请求实例的处理阶段，即“`$response = $kernel->handle();`”过程。请求的处理是服务器应用程序的核心功能，通过不同的处理方式最终返回形形色色的响应，实现不同的功能。如何提供可扩展的请求分发处理模块是服务器框架程序成功的关键，本书前面已经简单地介绍了路由和控制器，这只是请求处理过程中两个分点而已，下面将介绍整个处理的流程。具体源码如下：

文件 `Illuminate\Foundation\Http\Kernel.php`

```

// 处理一个输入 HTTP 请求
public function handle($request)
{
    try {
        $request->enableHttpMethodParameterOverride();
        $response = $this->sendRequestThroughRouter($request);
    }
    // 省略异常处理部分代码
    $this->app['events']->fire('kernel.handled', [$request, $response]);
    return $response;
}

```

请求的处理是通过 `sendRequestThroughRouter()` 方法实现的，通过该方法名就可以看出来，即通过路由传输请求实例。这里需要注意一点的是，`enableHttpMethodParameterOverride()` 方法会使能请求拒绝，被使能后在请求处理过程中会添加 CSRF 保护，即在客户端与服务端进行交互时，服务端会发送一个 CSRF 令牌给客户端，也就是一个 cookie，在客户端发送 POST 请求时需要将该令牌也发送给服务端，否则将拒绝处理该请求。

7.3.1 请求处理准备工作

在上一小节中，介绍了应用程序运行的准备环节，而要实现请求的处理，还有很多基础工作要做，根据 Laravel 框架的注释，将这个过程称为“拔靴带”过程，通俗的理解就像“拔靴带”一样，一个环节接一个环节进行处理。这里包括环境检测、配置加载、日记配置、异常处理、外观注册、服务提供者注册和启动服务共七个步骤，下面将对其中几个步骤的关键环节进行介绍。首先看一下这七个步骤是如何启动的，具体源码如下：

文件 `Illuminate\Foundation\Http\Kernel.php`

```
protected $bootstrappers = [
    'Illuminate\Foundation\Bootstrap\DetectEnvironment',
    'Illuminate\Foundation\Bootstrap\LoadConfiguration',
    'Illuminate\Foundation\Bootstrap\ConfigureLogging',
    'Illuminate\Foundation\Bootstrap\HandleExceptions',
    'Illuminate\Foundation\Bootstrap\RegisterFacades',
    'Illuminate\Foundation\Bootstrap\RegisterProviders',
    'Illuminate\Foundation\Bootstrap\BootProviders',
];

// 将请求通过中间件和路由处理
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);
    Facade::clearResolvedInstance('request');
    $this->bootstrap();
    return (new Pipeline($this->app))
        ->send($request)
        ->through($this->app->shouldSkipMiddleware() ? [] : $this->middleware)
        ->then($this->dispatchToRouter());
}

// 针对请求为应用程序“拔靴带”
public function bootstrap()
{
    if (! $this->app->hasBeenBootstrapped()) {
        $this->app->bootstrapWith($this->bootstrappers());
    }
}
```

文件 `Illuminate\Foundation\Application.php`

```
// 执行 bootstrap 类的数组
public function bootstrapWith(array $bootstrappers)
{
    $this->hasBeenBootstrapped = true;
```

```

foreach ($bootstrappers as $bootstrapper) {
    $this['events']->fire('bootstrapping: '.$bootstrapper, [$this]);
    $this->make($bootstrapper)->bootstrap($this);
    $this['events']->fire('bootstrapped: '.$bootstrapper, [$this]);
}
}

```

上面提到了，在请求处理的准备阶段共有七个环节，每一个环节是由一个类来负责实现的，而每个类都会有一个 `bootstrap()` 函数用于实现准备工作，这七个类名就存储在 `Illuminate\Foundation\Http\Kernel` 类的 `$bootstrappers` 数组属性中。在请求发送路由之前，首先通过 `bootstrap()` 函数来完成准备工作，该函数会调用服务容器实例中的 `bootstrapWith()` 函数，这里将通过代码 “`$this->make($bootstrapper)`” 完成每个准备类的实例化工作，然后调用准备类的 `bootstrap()` 方法实现准备工作。

1. 环境检测与配置加载

环境检测阶段是对程序运行的环境进行总体配置，这部分内容实际上和配置加载的功能是相同的，都是配置应用程序的运行环境，包括系统配置、身份认证配置、缓存配置、数据库配置、文件系统配置和 `sessions` 配置等。这些配置都是以文件形式提供的，其中环境检测文件是 Laravel 框架根目录下的 `.envn` 文件，而配置加载的配置文件是 `laravel\config\` 目录下的所有文件，两者的关系可以看做是主从的关系，即在配置加载过程中设置的参数都可以在 `.envn` 文件中进行设置，而 `.envn` 中对环境的配置将会覆盖配置加载项，当然也可以修改成不覆盖。说简单点，就是将一些重要的配置参数从 `laravel\config\` 目录下的文件中提取到 `.envn` 文件中，这样易于随时修改。首先给出环境检测加载的源码：

文件 `Illuminate\Foundation\Bootstrap\DetectEnvironment.php`

```

// 执行 bootstrap() 函数
public function bootstrap(Application $app)
{
    try {
        Dotenv::load($app->environmentPath(), $app->environmentFile());
    } catch (InvalidArgumentException $e) {
    }
    $app->detectEnvironment(function () {
        return env('APP_ENV', 'production');
    });
}

```

文件 `laravel\vendor\vlucas\phpdotenv\src\Dotenv.php`

```

// 在指定目录下加载 .env 文件
public static function load($path, $file = '.env')

```

```

{
    if (!is_string($file)) {
        $file = '.env';
    }
    $filePath = rtrim($path, '/') . '/' . $file;
    // 省略异常处理部分代码
    // 通过自动检测行结尾来读取文件每一行到数组中
    $autodetect = ini_get('auto_detect_line_endings');
    ini_set('auto_detect_line_endings', '1');
    $lines = file($filePath, FILE_IGNORE_NEW_LINES | FILE_SKIP_EMPTY_
LINES);

    ini_set('auto_detect_line_endings', $autodetect);
    foreach ($lines as $line) {
        if (strpos(trim($line), '#') === 0) {
            continue;
        }
        if (strpos($line, '=') !== false) {
            static::setEnvironmentVariable($line);
        }
    }
}

public static function setEnvironmentVariable($name, $value = null)
{
    list($name, $value) = static::normaliseEnvironmentVariable($name,
$value);
    if (static::$immutable === true && !is_null(static::findEnvironmentVariabl
e($name))) {
        return;
    }
    putenv("$name=$value");
    $_ENV[$name] = $value;
    $_SERVER[$name] = $value;
}

```

DetectEnvironment 类的 bootstrap() 函数通过 Dotenv::load() 静态函数实现 .env 文件的配置加载, 在 .env 文件中配置项以“配置项 = 参数值”的形式给出, 最后通过静态函数 setEnvironmentVariable() 给配置项“putenv("\$name=\$value");”设置环境变量, 并在 \$_ENV 和 \$_SERVER 全局数组中记录。

对于配置加载, 是通过 LoadConfiguration 类的 bootstrap() 函数实现的。部分源代码如下:

文件 Illuminate\Foundation\Bootstrap\LoadConfiguration.php

// 加载配置文件

```
public function bootstrap(Application $app)
```



```

{
    $items = [];
    if (file_exists($cached = $app->getCachedConfigPath())) {
        $items = require $cached;
        $loadedFromCache = true;
    }
    $app->instance('config', $config = new Repository($items));
    if (!isset($loadedFromCache)) {
        $this->loadConfigurationFiles($app, $config);
    }
    date_default_timezone_set($config['app.timezone']);
    mb_internal_encoding('UTF-8');
}

```

首先会查找是否有缓存的配置文件，如果有将先加载，这样加载配置项速度快，否则将文件顺序加载。对于程序配置项，将会存放到一个仓库类 (Repository 类) 实例中，而该类的实例被添加进服务容器的共享实例数组中，服务名称为“config”，以后就可以用该名称通过服务容器自动获得需要的配置参数。在完成仓库类的实例化和服务绑定后，将通过 loadConfigurationFiles() 函数进行配置项的加载。接下来介绍加载的实现过程。

文件 Illuminate\Foundation\Bootstrap\LoadConfiguration.php

```

// 加载所有配置文件的配置项
protected function loadConfigurationFiles(Application $app,
RepositoryContract $config)
{
    foreach ($this->getConfigurationFiles($app) as $key => $path) {
        $config->set($key, require $path);
    }
}
// 获取应用程序的所有配置文件
protected function getConfigurationFiles(Application $app)
{
    $files = [];
    foreach (Finder::create()->files()->name('*.php')->in($app->configPath()) as $file) {
        $nesting = $this->getConfigurationNesting($file);
        $files[$nesting.basename($file->getRealPath(), '.php')] = $file->getRealPath();
    }
    return $files;
}

```

文件 Illuminate\Config\Repository.php

```
// 设置配置项
public function set($key, $value = null)
{
    if (is_array($key)) {
        foreach ($key as $innerKey => $innerValue) {
            Arr::set($this->items, $innerKey, $innerValue);
        }
    } else {
        Arr::set($this->items, $key, $value);
    }
}
```

对于配置项的加载，首先需要获取配置文件，通过 `getConfigurationFiles()` 将 Laravel 框架下的配置文件全部读取出来并存储到 `$files` 数组中返回，该过程是通过服务容器获取配置文件的路径（代码“`$app->configPath()`”），然后通过 Symfony 组件中的探测类（`Symfony\Component\Finder\Finder` 类）实现文件的识别，最后提取出文件名和文件路径并以关联数组的形式（如“`app`”=>“`D:\WWW\laravel\config\app.php`”）存储到 `$files` 数组中返回。对于配置文件，每个文件返回一个数组，这里通过“`require '文件路径'`”的形式获取配置项数组，最后通过仓库实例的 `set()` 函数添加到仓库中。

前面讲到，`.env` 文件中的配置项会覆盖配置文件中的配置项，其实是通过 `env('APP_DEBUG', false)` 函数实现的，`env` 函数是 `Illuminate\Foundation\helpers.php` 文件中定义的，该文件定义了一些全局函数，可以在其他文件中调用。`env` 函数先检测环境变量，如果该环境变量存在，则返回环境变量，如果不存在则用第二参数作为返回值。前面讲到，在环境检测过程中，配置项已经通过 `putenv()` 设置为环境变量。

这里加载配置的目的是为在后面的程序运行过程中经常用到这些配置项时，可以通过服务容器方便地获取这些配置，“`$app['config']['app.aliases']`”和“`$app->make('config')->get('app.aliases')`”两种方法都是获取“`laravel\config\app.php`”文件中键名为 `aliases` 的值，即外观别名数组，因为在 `Container` 类和 `Repository` 类中都实现了 `ArrayAccess`（数组访问）接口，而 `$app['config']` 相当于调用该实例的 `offsetGet($key)` 方法，具体读者可以看一下源码，这里就不再赘述了。通过这部分的讲解，我们对于后面很多类似的用法将不再陌生了。

2. 外观注册

接下来介绍一下外观注册，因为在后面很多地方都用到了外观别名（所谓外观别名，就是给类名起了一个简洁而方便应用的别名），通过外观别名调用对应实例的属性和方法。根据官方文档上的介绍，外观别名的主要目的是为了测试的方便，但在程序中很多地方也用到了外观别名，比如路由，读者可能还会对 `Route::get("路径", "响应函数")` 这样的用法感到困惑，会以为 `Route` 是一个类，但是却找不到一个这样的对应类，其实这就是通过外观别名实现的，通过本节读者将了解它的全部。我们先看看在注册过程中做了什么，下面是部分源码：

文件 Illuminate\Foundation\Bootstrap\RegisterFacades.php

```
public function bootstrap(Application $app)
{
    Facade::clearResolvedInstances();
    Facade::setFacadeApplication($app);
    AliasLoader::getInstance($app->make('config')->get('app.aliases'))->
register();
}
```

文件 Illuminate\Foundation\AliasLoader.php

// 创建一个别名加载的实例对象

```
public static function getInstance(array $aliases = [])
{
    if (is_null(static::$instance)) {
        return static::$instance = new static($aliases);
    }
    $aliases = array_merge(static::$instance->getAliases(), $aliases);
    static::$instance->setAliases($aliases);
    return static::$instance;
}
```

同样，外观注册是通过 RegisterFacades 类的 bootstrap() 函数完成的，而外观注册也可以分为两个步骤来介绍，一是完成外观自动加载类的实例化并将外观别名数组添加到该实例中，这里需要与 composer 的自动加载类进行区别；二是完成外观自动加载类中的自动加载函数的添加。对于外观自动加载类 (AliasLoader 类) 的实例化是通过类的静态函数 getInstance() 实现的，而外观别名是在实例化过程中通过参数传入的，外观别名通过代码 “\$app->make('config')->get('app.aliases')” 获取，这句代码就用到了配置加载的内容，即获取加载的 app.php 配置文件，返回数组中的键值为 “aliases” 的数组值。可以通过源码查看一下该配置文件，了解一下都注册了哪些外观别名，由于篇幅原因，这里就不列举了。

// 在自动加载栈中注册一个自动加载函数

```
public function register()
{
    if (!$this->registered) {
        $this->prependToLoaderStack();
        $this->registered = true;
    }
}
// 将这个加载函数加到自动加载栈的开始处
protected function prependToLoaderStack()
{
    spl_autoload_register([$this, 'load'], true, true);
}
```



```
// 加载一个类的别名
public function load($alias)
{
    if (isset($this->aliases[$alias])) {
        return class_alias($this->aliases[$alias], $alias);
    }
}
```

在完成自动加载类的实例化后，会调用 `register()` 函数向类自动加载栈的开始处加入一个新的加载函数，即通过代码 “`spl_autoload_register([$this, 'load'], true, true);`” 实现。需要说明的是，实际上 PHP 的自动加载函数除了魔术方法 `__autoload()` 可以使用外，还可以在自动加载函数堆栈中添加自定义的自动加载函数，composer 提供的自动加载方法和外观自动加载方法都注册在这个堆栈中，而且外观自动加载在最前面。当对一个类进行自动加载时，要按照自动加载堆栈中注册的函数顺序完成自动加载，所以，所有类的自动加载都会先经过外观自动加载函数的处理，这个函数就是 `AliasLoader` 类实例的 `load()` 函数，该函数的作用是为外观类设置一个别名，即外观别名，当用户通过外观别名访问类时，实际上访问的是对应的外观类。

需要说明的是，对于别名 Laravel 框架中有两个，一个是容器核心别名，定义在 `Application` 类中，而存储在 `Application` 类实例的 `$aliases` 属性中（实际上该属性是在 `Container` 类中，因为 `Application` 类继承 `Container` 类，所以继承了这个属性，这里没有给出加上命名空间的全部名称）；另一个是外观别名，定义在 `app.php` 配置文件中，程序运行后存储在 `AliasLoader` 类实例的 `$aliases` 属性中。

下面我们看看通过外观别名是如何实现类似于 `Route::get(" 路径 ", " 响应函数 ")` 这样的函数调用的。对于这种函数的处理，程序首先会加载类 `Route`，由于注册了外观别名，那么自动加载栈的第一个函数是 `AliasLoader` 类的 `load()` 函数，而该函数会查找外观别名对应的类名，于是找到 `Route` 类的别名为 `Illuminate\Support\Facades\Route` 类，于是加载这个类，然后调用该类的静态方法 `get()`，而这个类没有这个静态方法。同时，对于外观类（在空间 `Facades` 下的类）都继承自 `Illuminate\Support\Facades\Facade` 外观父类，所以从这个父类中查找这个静态函数，而父类中也没有对应的静态函数，但是却有一个 `__callStatic()` 魔术方法，所以找不到静态方法时会调用这个魔术方法，接下来会调用一个 `getFacadeAccessor()` 静态方法，每一个具体的外观类都需要实现这个静态方法，该方法的目的是返回别名类所对应的在服务容器中服务的名称，对于 `Illuminate\Support\Facades\Route` 类来说，返回的是 “router”，接着通过服务容器获取对应的实例对象，这里对应的是 `Illuminate\Routing\Router` 类的实例，即通过 “`static::$app[$name]`” 实现，最终将会调用这个 `Router` 类实例中的 `get()` 方法。绕了一圈，终于绕出来了，现在明白注册外观的真正用途了吧，后面的如 `Request::input()` 等都是通过外观别名完成最终函数调用的。

3. 服务提供者注册

服务提供者注册为应用程序运行提供服务支持，在应用程序启动的准备阶段进行了基础服务提供者的加载，但这些服务只能应对前期启动阶段，而对于后期请求处理需要用到的数据库服务、认证服务、session 服务等还远远不够，所以这里会绑定后期使用的服务。首先给出部分源码：

文件 Illuminate\Foundation\Bootstrap\RegisterProviders.php

// 服务提供者注册

```
public function bootstrap(Application $app)
{
    $app->registerConfiguredProviders();
}
```

文件 Illuminate\Foundation\Application.php

// 注册所有配置的服务提供者

```
public function registerConfiguredProviders()
{
    $manifestPath = $this->getCacheServicesPath();
    (new ProviderRepository($this, new Filesystem, $manifestPath))
        ->load($this->config['app.providers']);
}
```

在服务提供者的注册过程中将服务提供者分为三类，即 when 类、eager 类和 deferred 类。when 类是注册事件，只有当事件发生时才会自动注册这个服务提供者；eager 类会直接加载，加载方式和注册基础服务提供者的过程相同；deferred 类的服务提供者存储在列表中，需要加载时才会加载。在 Laravel 框架中，提供了一个文件记录服务提供者的类别信息，即“laravel/bootstrap/cache/services.json;”，而该路径是通过 getCacheServicesPath() 函数获取的。服务提供者的注册经过两个步骤，第一步是服务提供者仓库的创建，即 registerConfiguredProviders() 函数中的 new ProviderRepository() 过程，第二步是通过 load() 函数进行注册。创建服务提供者仓库时需要提供该文件的路径参数，同时调用 load() 函数时会添加 app.php 配置文件中关于服务提供者的内容。那么两者之间是什么关系呢？通过下面的代码进行介绍。

文件 Illuminate\Foundation\ProviderRepository.php

// 注册应用的服务提供者

```
public function load(array $providers)
{
    // 加载 laravel\bootstrap\cache\services.json 文件中的服务提供者
    $manifest = $this->loadManifest();
    // 加载服务清单，这里包含程序所有的服务提供者并进行了分类
```

```

        if ($this->shouldRecompile($manifest, $providers)) {
            $manifest = $this->compileManifest($providers);
        }
        // 服务提供者加载事件，当这个事件发生时才自动加载这个服务提供者
        foreach ($manifest['when'] as $provider => $events) {
            $this->registerLoadEvents($provider, $events);
        }
        // 提前注册那些必须加载的服务提供者，以此为应用提供服务
        foreach ($manifest['eager'] as $provider) {
            $this->app->register($this->createProvider($provider));
        }
        // 在列表中记录延迟加载的服务提供者，需要时再加载
        $this->app->setDeferredServices($manifest['deferred']);
    }

```

前面提到服务提供者的记录在两个地方存在，实际上在配置文件 `app.php` 的 “providers” 中存储的是应用程序运行过程中所有的服务提供者，但是对于服务提供者的类型没有进行划分。当进行服务提供者注册时，首先通过 `loadManifest()` 函数读取 “`larave\bootstrap\cache\services.json`” 文件，该文件不仅记录服务提供者，还对服务提供者进行了分类，即前面提到的三种类型，但是该文件是一个临时记录文件，记录的是上一次运行时服务提供者加载的情况，所以需要与当前程序的服务提供者进行比对，通过 `shouldRecompile()` 函数实现，如果临时文件 (`services.json` 文件) 记录的服务提供者与配置文件 (`app.php` 文件) 记录的相同，则直接使用临时文件中服务提供者的分类进行注册。对于 `when` 类服务提供者，通过 `registerLoadEvents()` 函数创建事件监听者，当事件发生时调用服务提供者的 `register()` 函数进行服务注册 (也可以称为服务绑定)。对于 `eager` 类服务提供者，则直接通过 `createProvider()` 函数实例化服务提供者并进行服务注册。对于 `deferred` 类服务提供者，则记录在服务容器实例的 `$deferredServices` 数组属性中，在使用服务容器的 `make()` 函数进行服务解析时，如果发现这个服务在延时服务数组中，则会注册这个服务提供者，再解析相应的服务。

服务提供者都继承于服务提供者基类 (`Illuminate\Support\ServiceProvider`)，该类是一个抽象类，其中定义一个抽象函数 `register()`，所以每个服务提供者都需要实现该函数，而该函数中实现了服务注册的内容，所以服务提供者注册就是实例化服务提供者并调用该实例的 `register()` 函数，将服务绑定到服务容器实例中。

4. 启动服务

准备阶段的最后一个步骤是启动服务，服务提供者必须要实现 `register()` 函数，还有一个 `boot()` 函数根据需要决定是否实现，主要用于启动服务，而该函数不是必须的，如果不实现会在父类中统一处理。而对于实现 `boot()` 函数的服务提供者，会通过 `BootProviders` 类进行统一管理调用。下面给出部分源码：

文件 Illuminate\Foundation\Bootstrap\BootProviders.php

```
public function bootstrap(Application $app)
{
    $app->boot();
}
```

对于服务提供者启动管理类 BootProviders 来说, 实现比较简单, 只是调用服务容器中的 boot() 函数就可以了, 因为服务提供者的注册信息都记录在服务容器中。下面给出服务容器启动服务提供者的代码:

文件 Illuminate\Foundation\Application.php

// 启动应用程序的服务提供者

```
public function boot()
{
    if ($this->booted) {
        return;
    }
    $this->fireAppCallbacks($this->bootingCallbacks);
    array_walk($this->serviceProviders, function ($p) {
        $this->bootProvider($p);
    });
    $this->booted = true;
    $this->fireAppCallbacks($this->bootedCallbacks);
}

// 启动给定的服务提供者
protected function bootProvider(ServiceProvider $provider)
{
    if (method_exists($provider, 'boot')) {
        return $this->call([$provider, 'boot']);
    }
}
```

从上面可以看到, 在 Laravel 应用程序的服务容器中保存了服务提供者的实例数组, 即 \$serviceProviders 属性。这里包含了服务容器实例化过程中注册的两个基础服务提供者及在服务提供者注册过程中注册的 eager 类服务提供者, 然后通过代码 “\$this->call([\$provider, 'boot']);” 调用 \$serviceProviders 属性中记录的每一个服务提供者实例的 boot() 函数, 该函数主要是对相应的服务进行初始化, 如对于数据库服务提供者 (Illuminate\Database\DatabaseServiceProvider 类) 来讲会建立数据库连接, 而对于路由服务提供者 (App\Providers\RouteServiceProvider) 来讲会建立路由表等。这里介绍一下路由表的建立, 数据库连接的建立会在后面数据库部分讲解。

文件 app\Providers\RouteServiceProvider.php:

```
// 实现路由的配置
public function boot(Router $router)
{
    parent::boot($router);
}
```

文件 Illuminate\Foundation\Support\Providers\RouteServiceProvider.php

// 启动应用的路由服务

```
public function boot(Router $router)
{
    $this->setRootControllerNamespace();
    if ($this->app->routesAreCached()) {
        $this->loadCachedRoutes();
    } else {
        $this->loadRoutes();
        $this->app->booted(function () use ($router) {
            $router->getRoutes()->refreshNameLookups();
        });
    }
}
```

// 加载应用的路由

```
protected function loadRoutes()
{
    $this->app->call([$this, 'map']);
}
```

在 Laravel 框架中，默认情况下路由配置服务提供者是最后一个注册并启动的，该类位于 “app\Providers\” 文件夹下，通过上面的代码可以看出，该类调用父类的 boot() 函数，父类会通过 loadRoutes() 函数调用本实例对象的 map() 函数，这里需要注意的是，通过服务容器实例的 call() 函数调用一个函数时会解决函数的依赖问题。本书前面介绍了服务容器解决类实例化时构造函数的依赖注入问题，实际上服务容器也可以解决函数调用的依赖注入问题，就是通过 call() 函数来实现的，其内部机制基本相同，也是通过反射机制来解决依赖的，这里就不展开介绍了。也就是说，路由配置服务提供者需要自己实现一个 map() 函数来实现路由配置。

文件 app\Providers\RouteServiceProvider.php

// 为应用定义路由信息

```
public function map(Router $router)
{
    $router->group(['namespace' => $this->namespace], function ($router) {
        require app_path('Http/routes.php');
    });
}
```

文件 Illuminate\Routing\Router.php

```
// 创建一个路由组
public function group(array $attributes, Closure $callback)
{
    $this->updateGroupStack($attributes);
    call_user_func($callback, $this);
    array_pop($this->groupStack);
}
```

文件 laravel\app\Http\routes.php (文件路径)

```
Route::controller('home', 'HomeController');
Route::get('/', WelcomeController@index');
```

上文提到了通过服务容器的 call() 函数调用其他实例的函数可以解决依赖注入的问题，对于 map() 函数，依赖一个参数为 Illuminate\Routing\Router 类的实例，会通过服务容器来解决这个依赖，而这个实例对象（以后可以将其称为路由器对象）管理应用程序的路由情况，所有的路由配置信息最终都会记录在路由器实例对象中。map() 函数会调用路由器实例的 group() 函数，该函数会记录控制器命名空间，默认情况下是“App\Http\Controllers”，在解析路由时遇到的控制器名会自动加上该命名空间进行解析。然后会调用一个回调函数，函数只有一句代码“require app_path('Http/routes.php');”，也就是包含并执行路由配置文件。前面已经介绍了路由的创建，这里通过 controller() 方法和 get() 方法定义了两个路由配置项，get() 方法会在路由器实例中生成一条路由信息，而 controller() 方法则会生成多条，与控制器中的方法相对应。下面进一步介绍路由信息是如何配置的。

文件 Illuminate\Routing\Router.php

```
// 根据 uri 的前缀配置控制器路由
public function controller($uri, $controller, $names = [])
{
    $prepending = $controller;
    if (! empty($this->groupStack)) {
        $prepending = $this->prependGroupUses($controller);
    }
    $routable = (new ControllerInspector)->getRoutable($prepending, $uri);
    foreach ($routable as $method => $routes) {
        foreach ($routes as $route) {
            $this->registerInspected($route, $controller, $method, $names);
        }
    }
    $this->addFallthroughRoute($controller, $uri);
}
// 注册一个控制器路由
protected function registerInspected($route, $controller, $method,
```



```

&$names)
{
    $action = ['uses' => $controller.'@'.$method];
    $action['as'] = Arr::get($names, $method);
    $this->{$route['verb']}($route['uri'], $action);
}
// 添加一条路由配置到路由集合表中
protected function addRoute($methods, $uri, $action)
{
    return $this->routes->add($this->createRoute($methods, $uri, $action));
}

```

通过 `controller()` 方法实现配置路由的过程要比 `get()` 方法复杂一些，首先会通过 `prependGroupUses()` 函数获取控制器的整个命名空间，上述实例得到的控制器类是 “App\Http\Controllers\HomeController”，然后会根据路由前缀（本实例中是 “home”）获取整个路由信息的配置数组，数组中每一项表示一个路由信息，包括请求方法和 URI 地址等，这些信息将逐项加入到路由集合中，即控制器实例的属性 `$routes`，而添加过程中不同的请求方法对应不同的函数，所以 `controller()` 方法最终也需要调用 `get()`、`post()` 等方法配置路由。在完成路由配置后，整个启动阶段就基本完成了。

这里我们只讲解了七个 “bootstrap” 类中的五个类，也就对应七个准备环节中的五个环节，对于日志配置和异常处理两个环节通过名字我们就可以了解其主要作用，即对日志的记录和对异常处理进行配置，这里就不详细介绍了。

7.3.2 中间件

在请求处理的过程中，经过烦琐的准备工作，应用程序开始对请求进行了处理。对请求的处理，Laravel 框架是逐级进行的，首先是经过中间件的处理，然后经过路由处理，最后到控制器生成响应，这部分思想在 6.2.1 节装饰者模式中进行了介绍，对请求从中间件到路由处理再到响应生成整个过程中基本是以装饰者模式的思想进行处理的。那么中间件会对请求做哪些处理呢？下面给出部分源码：

文件 `laravel\app\Http\Kernel.php`

```

protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\
    CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
];

```

文件 Illuminate\Foundation\Http\Kernel.php

// 将请求通过中间件和路由处理

```
protected function sendRequestThroughRouter($request)
{
    $this->app->instance('request', $request);
    Facade::clearResolvedInstance('request');
    $this->bootstrap();
    return (new Pipeline($this->app))
        ->send($request)
        ->through($this->app->shouldSkipMiddleware() ? [] : $this-
>middleware)
        ->then($this->dispatchToRouter());
}
```

// 设置路由分发回调函数

```
protected function dispatchToRouter()
{
    return function ($request) {
        $this->app->instance('request', $request);
        return $this->router->dispatch($request);
    };
}
```

文件 Illuminate\Pipeline\Pipeline.php

```
public function __construct(Container $container)
{
```

```
    $this->container = $container;
```

```
}
```

// 设置被送入“管道”的对象

```
public function send($passable)
```

```
{
```

```
    $this->passable = $passable;
```

```
    return $this;
```

```
}
```

// 设置导管数组

```
public function through($pipes)
```

```
{
```

```
    $this->pipes = is_array($pipes) ? $pipes : func_get_args();
```

```
    return $this;
```

```
}
```

// 以一个回调函数为终点执行“管道”处理

```
public function then(Closure $destination)
```

```
{
```

```
    $firstSlice = $this->getInitialSlice($destination);
```

```
    $pipes = array_reverse($this->pipes);
```

```

        return call_user_func(
            array_reduce($pipes, $this->getSlice(), $firstSlice), $this->
>passable
        );
    }
    // 获取一个用来代替应用处理“洋葱”层的回调函数
    protected function getSlice()
    {
        return function ($stack, $pipe) {
            return function ($passable) use ($stack, $pipe) {
                if ($pipe instanceof Closure) {
                    return call_user_func($pipe, $passable, $stack);
                } else {
                    list($name, $parameters) = $this->parsePipeString($pipe);
                    return call_user_func_array([$this->container->make
($name),$this->method], array_merge([$passable, $stack], $parameters));
                }
            };
        };
    }
}

```

在 Laravel 框架中，很多注释和代码名称已经非常形象地表达了程序代码的功能，代码注释中将中间件称为“洋葱”层，将整个处理流程称为“管道”，有些地方会用到这些名词，如果读者理解了真正的含义就会更容易理解程序。对请求的处理阶段，首先对管道类(Pipeline 类)进行了实例化，分别通过 `send()` 函数和 `through()` 函数将请求实例和中间件数组赋值给管道实例，而最终的处理是通过 `then()` 函数完成的，该函数有一个参数，这个参数是经过“管道”后的终点处理函数，即下一步的路由处理。而 `then()` 函数其实就是将整个中间件数组通过服务容器生成实例，并对这些实例的 `handle()` 函数和传入的终点处理回调函数进行组装，形成一个递归调用的回调函数，再进行调用，最终完成“管道”的逐级处理。如果对这部分代码理解有困难，请先阅读第6章中请求处理管道的内容。

对于初始的 Laravel 框架程序，中间件包括 `CheckForMaintenanceMode`、`EncryptCookies`、`AddQueuedCookiesToResponse`、`StartSession`、`ShareErrorsFromSession` 和 `VerifyCsrfToken` 共六类，定义在“`laravel\app\Http\Kernel.php`”文件中的 `$middleware` 数组中，分别用来完成验证维护模式、Cookie 加密、添加响应 Cookie、开启会话、共享 Session 错误和 CSRF 保护六个部分，其中一部分重要的内容将会在后面其他章节中进行详细介绍。如果在程序开发过程中需要添加新的中间件，可以按照这几个类进行设计并添加到中间件数组中。经过对这部分的了解，添加中间件应该非常简单了。

7.3.3 路由处理生成响应

对于 Laravel 框架，请求是通过路由与控制器的响应函数对应的，对于路由的设定在第 5 章“Laravel 框架初识”部分已经介绍过，是在“`laravel/app/Http/routes.php`”文件中定义的，那么这部分内容这么关键，为何在程序运行中没有看到呢？其实整个路由表的生成是在请求处理的准备工作中的启动服务过程中完成的，通过 `RouteServiceProvider` 中的 `boot()` 函数实现。下面将按请求处理的步骤逐步介绍，让读者对请求的处理过程有一个全面的了解。

1. 路由匹配

在路由的定义过程中，通常有两个参数，一个是 URI（唯一资源标识符），另一个是处理函数或处理函数定位，而大部分是处理函数定位，即控制器中相应的函数。那么，在路由表中就会请求形式与处理函数的对应信息，每一个对应信息为一个路由，通过 `Illuminate\Routing\Route` 实例进行保存，而所有的路由信息又通过 `Illuminate\Routing\RouteCollection` 实例保存形成路由表，而路由表则由 `Illuminate\Routing\Router` 类实例保存，这就是路由信息的结构。对于请求的处理，首先是在路由信息结构中找到对应的路由，即对应的 `Illuminate\Routing\Route` 实例，下面是这部分源码：

文件 `Illuminate\Foundation\Http\Kernel.php`

```
// 获取路由分发回调函数
protected function dispatchToRouter()
{
    return function($request)
    {
        $this->app->instance('request', $request);
        return $this->router->dispatch($request);
    };
}
```

上面提到，所有的路由信息其实都保存在一个 `Illuminate\Routing\Router` 类实例中，而这个类实例存储在 `Kernel` 类的实例当中，于是通过“`$this->router->dispatch($request);`”将请求信息传递给路由信息存储实例。

文件 `Illuminate\Routing\Router.php`

```
public function dispatch(Request $request)
{
    $this->currentRequest = $request;
    $response = $this->callFilter('before', $request);
    if (is_null($response))
    {
        $response = $this->dispatchToRoute($request);
    }
}
```

```

    }
    $response = $this->prepareResponse($request, $response);
    $this->callFilter('after', $request, $response);
    return $response;
}
// 分发请求到指定路由并返回一个响应
public function dispatchToRoute(Request $request)
{
    $route = $this->findRoute($request);
    $request->setRouteResolver(function() use ($route)
    {
        return $route;
    });
    $this->events->fire('router.matched', [$route, $request]);
    $response = $this->callRouteBefore($route, $request);
    if (is_null($response)) {
        $response = $this->runRouteWithinStack($route, $request);
    }
    $response = $this->prepareResponse($request, $response);
    $this->callRouteAfter($route, $request, $response);
    return $response;
}

// 通过实例栈启动给定的路由
protected function runRouteWithinStack(Route $route, Request $request)
{
    $middleware = $this->gatherRouteMiddlewares($route);
    return (new Pipeline($this->container))
        ->send($request)
        ->through($middleware)
        ->then(function($request) use ($route)
        {
            return $this->prepareResponse(
                $request, $route->run($request));
        });
}

```

在路由信息存储实例中，通过“`$route = $this->findRoute($request);`”来查找请求对应的路由实例，查找主要是根据请求的方法和请求 URI 来实现对应，当查找到请求对应的路由后，请求将会传递到对应的路由中去处理，即“`$route->run($request)`”。

文件 `Illuminate\Routing\Route.php`

```

// 执行路由动作并返回响应
public function run(Request $request)

```

```

    {
        $this->container = $this->container ?: new Container;
        try {
            if (!is_string($this->action['uses'])) {
                return $this->runCallable($request);
            }
            if ($this->customDispatcherIsBound()) {
                return $this->runWithCustomDispatcher($request);
            }
            return $this->runController($request);
        } catch (HttpResponseException $e) {
            return $e->getResponse();
        }
    }
}

// 将请求发送到常规分发器去处理
protected function runWithCustomDispatcher(Request $request)
{
    list($class, $method) = explode('@', $this->action['uses']);
    $dispatcher = $this->container->make('illuminate.route.
dispatcher');
    return $dispatcher->dispatch($this, $request, $class, $method);
}

```

在请求对应的路由中，会检测是否使用常规的控制分发器去处理，在初始的 Laravel 框架中使用的是常规控制分发器，通过服务容器自动生成这个控制分发器，这个服务是通过服务提供者 `ControllerServiceProvider` 注册的，下一步将会把请求及路由中关于处理函数的信息交给控制分发器去处理，这里对路由中关于处理函数的信息是以控制器类名和函数名给出的，即 `$class` 和 `$method`。

2. 控制器生成

在控制分发器中，将会根据路由提供的响应函数信息来实例化控制器类，并调用对应的响应函数生成响应的内容部分。下面给出部分源码：

文件 `Illuminate\Routing\ControllerDispatcher.php`

```

// 分发一个请求到一个给定的控制器中的方法
public function dispatch(Route $route, Request $request, $controller,
$method)
{
    $instance = $this->makeController($controller);
    $this->assignAfter($instance, $route, $request, $method);
    $response = $this->before($instance, $route, $request, $method);
    if (is_null($response))
    {

```



```

        $response = $this->callWithinStack($instance, $route, $request,
$method);
    }
    return $response;
}
// 通过服务容器创建控制器实例
protected function makeController($controller)
{
    Controller::setRouter($this->router);
    return $this->container->make($controller);
}
// 调用给定的控制器实例的对应方法
protected function callWithinStack($instance, $route, $request, $method)
{
    $middleware = $this->getMiddleware($instance, $method);
    return (new Pipeline($this->container))
        ->send($request)
        ->through($middleware)
        ->then(function($request) use ($instance, $route, $method){
            return $this->router->prepareResponse($request,
$this->call($instance, $route, $method) );
        });
}
// 调用给定的控制器实例的方法
protected function call($instance, $route, $method)
{
    $parameters = $this->resolveClassMethodDependencies(
        $route->parametersWithoutNulls(), $instance, $method
    );
    return $instance->callAction($method, $parameters);
}

```

在控制分发器中，首先根据控制器类名，通过服务容器进行实例化，再通过调用控制器实例对应的方法 (`$instance->callAction($method, $parameters)`) 来生成响应的主体部分。这里需要注意，请求并不是都能正常地到达处理函数生成响应，因为还有权限等问题，那么 Laravel 是如何对请求的权限等信息进行控制的呢？这是中间件发挥了作用。在控制器的构造函数中，可以通过 “`$this->middleware('auth');`” 来声明该控制器中处理函数在处理请求过程中需要进行对应中间件的处理，这里当然也可以指定哪些处理函数在响应请求时需要经过中间件。当声明了中间件后，就会在 `callWithinStack()` 函数中获取对应的中间件信息，即 “`$middleware = $this->getMiddleware($instance, $method);`”，然后通过“管道”方式进行处理。在 Laravel 框架中，对于身份验证等功能就是这样实现的，在响应之前首先经过 “auth” 中间件（在 `laravel/app/Http/Kernel.php` 文件中的 `$routeMiddleware` 数组中定义）的 `handle()`

函数进行处理，如果验证通过将会进行响应处理，否则重定向到登录页面，这部分内容在后面的章节中还会详细介绍。

3. 响应生成

经历了层层的处理，应用程序终于找到了对应请求的处理函数，这里假设就是 Laravel 框架的欢迎页面。对于不同的应用可能生成的响应主体是不同的，对于移动应用可能响应的就是一些 JSON 格式的数据，而对于网页应用响应的可能是一个 HTML 页面。对于 HTML 页面，在第 5 章中的视图部分进行了介绍，但是没有涉及生成的过程，在这里将分析这个生成过程。下面是部分源码：

文件 `laravel\app\Http\Controllers\WelcomeController.php`

```
// 显示视图到屏幕
public function index()
{
    return view('welcome');
}
```

文件 `Illuminate\Foundation\helpers.php`

```
// 根据给定的视图名称得到视图内容
if ( ! function_exists('view'))
{
    function view($view = null, $data = array(), $mergeData = array())
    {
        $factory = app('Illuminate\Contracts\View\Factory');
        if (func_num_args() === 0)
        {
            return $factory;
        }
        return $factory->make($view, $data, $mergeData);
    }
}
```

文件 `Illuminate\View\Factory.php`

```
// 生成视图实例
public function make($view, $data = array(), $mergeData = array())
{
    if (isset($this->aliases[$view])) $view = $this->aliases[$view];
    $view = $this->normalizeName($view);
    $path = $this->finder->find($view);
    $data = array_merge($mergeData, $this->parseData($data));
    $this->callCreator($view = new View($this, $this->
```

```
>getEngineFromPath($path), $view, $path, $data));
    return $view;
}
```

对于视图的生成实际上是通过实例化 `Illuminate\View\View` 类实现的，在 `View` 实例中包含了视图文件路径、名称、数据及它的编译引擎等，接下来将会根据生成的响应主体生成响应实例。在 `Laravel` 框架中对响应的封装是通过 `Illuminate\Http\Response` 类完成的，而该类的底层也用到了 `Symfony` 框架中的 `Response` 类，即 `Symfony\Component\HttpFoundation\Response` 类。

文件 `Illuminate\Routing/Router.php`

// 根据给定的内容创建响应实例

```
protected function prepareResponse($request, $response)
{
    if ( ! $response instanceof SymfonyResponse)
    {
        $response = new Response($response);
    }
    return $response->prepare($request);
}
```

文件 `laravel/vendor/symfony/http-foundation/Response.php`

// 在响应传输到客户端之前进行准备

```
public function prepare(Request $request)
{
    $headers = $this->headers;
    if ($this->isInformational() || $this->isEmpty()) {
        $this->setContent(null);
        $headers->remove('Content-Type');
        $headers->remove('Content-Length');
    } else {
        if (!$headers->has('Content-Type')) {
            $format = $request->getRequestFormat();
            if (null !== $format && $mimeType = $request->getMimeType($format)) {
                $headers->set('Content-Type', $mimeType);
            }
        }
        $charset = $this->charset ? 'UTF-8';
        if (!$headers->has('Content-Type')) {
            $headers->set('Content-Type', 'text/html; charset='.$charset);
        } elseif (0 === strpos($headers->get('Content-Type'),
```



```

'text/') && false === stripos($headers->get('Content-Type'), 'charset')) {
    $headers->set('Content-Type', $headers->get('Content-
Type').'; charset='.$charset);
}
if ($headers->has('Transfer-Encoding')) {
    $headers->remove('Content-Length');
}
if ($request->isMethod('HEAD')) {
    $length = $headers->get('Content-Length');
    $this->setContent(null);
    if ($length) {
        $headers->set('Content-Length', $length);
    }
}
}
if ('HTTP/1.0' != $request->server->get('SERVER_PROTOCOL')) {
    $this->setProtocolVersion('1.1');
}
if ('1.0' == $this->getProtocolVersion() && 'no-cache' == $this-
>headers->get('Cache-Control')) {
    $this->headers->set('pragma', 'no-cache');
    $this->headers->set('expires', -1);
}
$this->ensureIEOverSSLCompatibility($request);
return $this;
}

```

从上面可以看到，响应最终是封装在 `Illuminate\Http\Response` 实例中的，其中不仅包括了控制器处理函数得到的响应主体，还包括根据请求生成的响应头的内容。至此，从请求的发出到响应的生成就已经完成了，接下来就是将响应发给客户端并记录相关的信息。

7.4 响应的发送与程序终止

7.4.1 响应的发送

到这里，对 HTTP 请求的响应已经生成了，接下来需要将封装在 `Illuminate\Http\Response` 实例中的响应以 HTTP 响应的形式发送给客户端，实现一个请求生命周期的最后环节。响应的发送是在 `laravel\public\index.php` 文件中通过 “`$response->send();`” 实现的。

文件 `laravel\public\index.php`

```
$response->send();
```

文件 `laravel/vendor/symfony/http-foundation/Response.php`

```
// 发送 HTTP 响应头和内容
public function send()
{
    $this->sendHeaders();
    $this->sendContent();
    if (function_exists('fastcgi_finish_request')) {
        fastcgi_finish_request();
    } elseif ('cli' !== PHP_SAPI) {
        static::closeOutputBuffers(0, true);
    }
    return $this;
}

// 发送 HTTP 头部内容
public function sendHeaders()
{
    if (headers_sent()) {
        return $this;
    }
    header(sprintf('HTTP/%s %s %s', $this->version, $this->statusCode,
    $this->statusText), true, $this->statusCode);
    foreach ($this->headers->allPreserveCase() as $name => $values) {
        foreach ($values as $value) {
            header($name.': '.$value, false, $this->statusCode);
        }
    }
    foreach ($this->headers->getCookies() as $cookie) {
        setcookie($cookie->getName(), $cookie->getValue(), $cookie-
        >getExpiresTime(), $cookie->getPath(), $cookie->getDomain(), $cookie-
        >isSecure(), $cookie->isHttpOnly());
    }
    return $this;
}

// 发送 Web 响应的内容
public function sendContent()
{
    echo $this->content;
    return $this;
}
```

响应的发送包括两部分内容，分别是响应头信息的发送和响应主体内容的发送。响应头信息包括状态行、首部字段和 Cookie 的发送，状态行和首部字段是通过 `header()` 函数完成的，Cookie 的发送是通过 `setcookie()` 函数完成的，这里的 Cookie 内容主要是 session 的

ID 及 CSRF（跨网站请求伪造）令牌，一个用于会话控制，另一个是防止 CSRF 攻击。在响应发送完成后，通过调用 `closeOutputBuffers()` 静态函数完成缓冲区的释放。

7.4.2 程序终止

在完成 HTTP 响应的发送后，接下来进入程序生命周期的最后阶段——程序终止，对于 Laravel 框架，程序终止主要是完成终止中间件的调用。

文件 `laravel\public\index.php`

```
$kernel->terminate($request, $response);
```

文件 `Illuminate\Foundation\Http\Kernel.php`

// 调用终止中间件的 `terminate` 方法

```
public function terminate($request, $response)
{
    $middlewares = $this->app->shouldSkipMiddleware() ? [] : array_merge(
        $this->gatherRouteMiddlewares($request),
        $this->middleware
    );
    foreach ($middlewares as $middleware) {
        list($name, $parameters) = $this->parseMiddleware($middleware);
        $instance = $this->app->make($name);
        if (method_exists($instance, 'terminate')) {
            $instance->terminate($request, $response);
        }
    }
    $this->app->terminate();
}
```

中间件的内容在前面小节已经介绍过，在“`App\Http\Kernel`”类的 `$middleware` 数组属性中进行管理，默认情况下，需要终止的中间件就是指这个数组中记录的中间件，然后通过服务容器依次实例化相应的中间件并调用 `terminate()` 函数。在 Laravel 框架默认情况下，只有会话中间件存在该函数，主要用于会话记录，即用户状态信息的记录，这部分内容会在第 12 章中介绍，这里只需要了解在程序终止阶段还会调用中间件，如果有需求可以在这部分添加相应的功能。

本章介绍了请求到响应的整个执行过程，主要可以归纳为四个阶段，即程序启动准备阶段、请求实例化阶段、请求处理阶段、响应发送和程序终止阶段。每个阶段都有相应的职责功能。程序启动准备阶段主要完成文件自动加载的实现、服务容器的实例化、基础服务提供者的注册及核心类的实例化等，核心类实例对象用于控制请求实例对象生成和处理过程的各个环节，而服务容器实例化是为整个过程提供资源服务。请求实例化阶段是将请求信息以

对象的形式进行记录保存的过程。请求处理阶段首先是准备请求处理的环境，包括环境加载、服务提供者注册等七个环节，然后将请求实例通过中间件处理及通过路由和控制器的分发控制，使得不同请求通过相应的处理函数进行处理并生成响应的过程。响应发送和程序终止阶段是将响应返回给客户端并记录与客户端有关的信息等工作。这就是 Laravel 框架的整个生命周期过程。

请求到响应的生命周期

1.1 请求到响应的生命周期

请求到响应的生命周期是指从用户发起请求开始，到服务器接收到请求，经过一系列的处理，最终将响应返回给客户端的过程。这个过程可以分为以下几个阶段：

1. 请求接收：当客户端发送请求到服务器时，服务器会接收到这个请求。在 Laravel 中，这个请求会被封装成一个 `Illuminate\Http\Request` 对象，并传递给 `Illuminate\Routing` 组件。

2. 路由分发：路由分发是请求处理的第一步。它负责将请求分发到相应的控制器。在 Laravel 中，路由分发是由 `Illuminate\Routing` 组件负责的。它会检查请求的 URL、HTTP 方法等信息，并尝试匹配路由规则。一旦匹配成功，它会将请求分发到相应的控制器。

3. 中间件处理：中间件是请求处理过程中的一个重要环节。它可以在请求到达控制器之前或之后执行一些操作。在 Laravel 中，中间件是由 `Illuminate\Pipeline` 组件负责的。它可以对请求进行过滤、修改或拒绝。例如，你可以使用中间件来验证用户身份、检查请求是否来自允许的 IP 地址等。

4. 控制器处理：控制器是请求处理的核心。它负责接收请求，执行业务逻辑，并返回响应。在 Laravel 中，控制器是由 `Illuminate\Routing` 组件分发的。你可以定义自己的控制器，并让路由规则将请求分发到它们。

5. 响应生成：控制器处理完请求后，会返回一个响应对象。这个响应对象可以是字符串、视图、JSON 数据等。在 Laravel 中，响应生成是由 `Illuminate\Routing` 组件负责的。它会将控制器的返回结果封装成一个 `Illuminate\Http\Response` 对象。

6. 响应发送：最后，服务器会将响应对象发送给客户端。在 Laravel 中，这个步骤是由 `Illuminate\Http` 组件负责的。它会负责将响应对象转换成 HTTP 响应格式，并发送给客户端。

请求到响应的生命周期 1.8

请求到响应的生命周期是指从用户发起请求开始，到服务器接收到请求，经过一系列的处理，最终将响应返回给客户端的过程。这个过程可以分为以下几个阶段：

1. 请求接收：当客户端发送请求到服务器时，服务器会接收到这个请求。在 Laravel 中，这个请求会被封装成一个 `Illuminate\Http\Request` 对象，并传递给 `Illuminate\Routing` 组件。

2. 路由分发：路由分发是请求处理的第一步。它负责将请求分发到相应的控制器。在 Laravel 中，路由分发是由 `Illuminate\Routing` 组件负责的。它会检查请求的 URL、HTTP 方法等信息，并尝试匹配路由规则。一旦匹配成功，它会将请求分发到相应的控制器。

3. 中间件处理：中间件是请求处理过程中的一个重要环节。它可以在请求到达控制器之前或之后执行一些操作。在 Laravel 中，中间件是由 `Illuminate\Pipeline` 组件负责的。它可以对请求进行过滤、修改或拒绝。例如，你可以使用中间件来验证用户身份、检查请求是否来自允许的 IP 地址等。

4. 控制器处理：控制器是请求处理的核心。它负责接收请求，执行业务逻辑，并返回响应。在 Laravel 中，控制器是由 `Illuminate\Routing` 组件分发的。你可以定义自己的控制器，并让路由规则将请求分发到它们。

5. 响应生成：控制器处理完请求后，会返回一个响应对象。这个响应对象可以是字符串、视图、JSON 数据等。在 Laravel 中，响应生成是由 `Illuminate\Routing` 组件负责的。它会将控制器的返回结果封装成一个 `Illuminate\Http\Response` 对象。

6. 响应发送：最后，服务器会将响应对象发送给客户端。在 Laravel 中，这个步骤是由 `Illuminate\Http` 组件负责的。它会负责将响应对象转换成 HTTP 响应格式，并发送给客户端。

第 8 章

服务容器与服务提供者

在 Laravel 框架中，如果问什么是设计最巧妙也是最需要掌握的内容，那么一定就是服务容器和服务提供者。Laravel 框架之所有能够具备低耦合、易扩展和可重用的优秀特性，也正是因为服务容器和服务提供者。如果将整个 Laravel 框架比喻成一个人，那么服务容器就相当于人的大脑，而服务提供者相当于神经系统，各个功能组件相当于人的手、脚、耳、鼻等功能部位，其中服务容器掌控着所有框架的功能信息，相当于人脑掌控着身体的所有功能器官，而人脑掌控人的各个功能器官是通过神经系统实现的，Laravel 框架中的服务提供者就扮演着这样的角色。服务提供者首先需要将各个功能模块具备的功能注册到服务容器中，当需要完成某些功能时，服务容器会通过服务提供者注册的服务完成相应的准备，如功能模块的实例化及参数配置等，然后会调用准备好的功能模块实现相应的功能。在 Laravel 框架中，服务容器只有一个，相当于人的一个大脑，而服务提供者遍布整个框架的各个功能模块内。服务容器的实现在 Laravel 框架的设计模式中已经进行了介绍，这里将着重介绍它的应用及如何与服务提供者实现协调工作。

8.1 服务容器

服务容器的概念如果从程序的角度去理解很难知道它是用来做什么的，其实只需要从字面上理解就好，容器即为装东西的器皿，而服务容器其实就是用来装服务的。对于程序来说什么是服务？任何程序中用到的都可以当成服务，如类、实例、文件路径等，Laravel 框架的服务容器就是用来装这些东西的。下面将介绍服务容器的使用。

8.1.1 服务容器的产生

在 Laravel 框架中，服务容器是由 `Illuminate\Container\Container` 类实现的，该类实现了服务容器的核心功能，而 `Illuminate\Foundation\Application` 类继承了该类，主要实现了服务容器的初始配置和功能扩展。要使用服务容器，首先需要产生服务容器，对于 Laravel 框架，当接收到一个请求时，就会为了处理这个请求首先生成一个服务容器，用于容纳处理请求需

要的服务。实现过程代码如下：

文件 `laravel\public\index.php`

```
<?php
// 注册自动加载文件
require __DIR__.'../../bootstrap/autoload.php';
// 服务容器生成
$app = require_once __DIR__.'../../bootstrap/app.php';
```

文件 `laravel\bootstrap\app.php`

```
<?php
// 服务容器创建
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'../../')
);
$app->singleton(
    Illuminate\Contracts\Http\Kernel::class,
    App\Http\Kernel::class
);
// 省略命令行核心类绑定和异常类绑定的代码
return $app;
```

通过上述两个文件中的代码就实现了服务容器的实例化生成，其中 `index.php` 是请求的入口文件，当程序接收到一个请求时，首先会加载 `bootstrap` 目录下的 `autoload.php` 和 `app.php` 文件，用笔者的说法就是首先要点亮程序应用，其中 `autoload.php` 文件用于加载 `autoload`，有了它就可以根据命名空间找到相应类文件的位置并实现自动加载，而 `app.php` 文件实现了服务容器的实例化，同时绑定了核心处理类。至此，已经获得了一个全局的服务容器实例，即 `$app`。

8.1.2 服务绑定

在实现了服务容器的生成后，首先要做的就是向其中装服务，也就是所谓的容器绑定。这里一直说绑定，究竟是什么和什么进行绑定呢？实际上可以简单地理解为一个服务和一个关键字进行绑定，可以简单看做是一种键值对形式，即一个“key”对应一个服务。对于绑定服务的不同，需要服务容器中不同的绑定函数来实现，主要包括回调函数服务绑定和实例对象服务绑定。回调函数服务绑定的就是一个回调函数，而实例对象服务绑定的是一个实例对象。回调函数的绑定还分为两种，一种是普通绑定，另一种是单例绑定。普通绑定每次生成该服务的实例对象时都会生成一个新的实例对象，也就是说在程序的生命周期中，可以同时生成很多个这种实例对象，而单例绑定在生成一个实例对象后，如果再次生成就会

返回第一次生成的实例对象，也就是说在程序的生命周期中，只能生成一个这样的实例对象，如果想使用就会获取之前生成的，也就是设计模式中的单例模式。具体实现代码如下：

文件 `laravel\app\ServiceTest\GeneralService.php`

```
<?php    namespace App\ServiceTest;
class GeneralService
{
    public $serviceName;
}
```

文件 `laravel\app\ServiceTest\SingleService.php`

```
<?php    namespace App\ServiceTest;
class SingleService
{
    public $serviceName;
}
```

文件 `laravel\app\ServiceTest\InstanceService.php`

```
<?php    namespace App\ServiceTest;
class InstanceService
{
    public $serviceName;
}
```

文件 `laravel\bootstrap\app.php`

```
<?php
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../')
);
$app->bind(App\ServiceTest\GeneralService::class, function($app){
    return new App\ServiceTest\GeneralService();
});
$app->singleton(App\ServiceTest\SingleService::class, function($app){
    return new App\ServiceTest\SingleService();
});
$instance = new App\ServiceTest\InstanceService();
$app->instance('instanceService', $instance);
```

在上述实例代码中，创建了三个简单的类，可以假设三个类代表了三种不同的功能。在 `bootstrap\app.php` 文件中，当服务容器实例刚刚创建完成后即通过三种不同的方式绑定服务，一种是普通模式绑定回调函数，另一种是单例模式绑定回调函数，还有一种是绑定一个实例对象。那么在实现三种形式的绑定后，在服务容器中增加了如下内容：

```
$bindings = array( "App\ServiceTest\GeneralService" =>array( "concrete" =>{
Closure}, "shared" =>false), "App\ServiceTest\SingleService" =>array( "concret
e" =>{Closure}, "share" =>true));
$instances = array( "instanceService" =>{App\ServiceTest\InstanceService});
```

这里可以看出，服务容器是通过 \$bindings 属性和 \$instances 属性来记录本实例中绑定的服务的，上面列出了增加的内容，实际上在服务容器初始化的过程中就绑定了一些服务。对于回调函数服务绑定是在 \$bindings 中记录的，其键即为绑定的服务名称，值是回调函数和模式标识，如果是普通模式则 “share” 值为 false，如果是单例模式则该值为 true。实例对象服务绑定是在 \$instances 中记录的，键为服务名称，值为相应类的实例对象。

除了上述直接绑定回调函数和实例对象的方式，还有一种形式的绑定，即绑定具体类名称，本质上也是绑定回调函数的方式，只是回调函数是服务容器根据提供的参数自动生成的。实例代码如下：

文件 laravel\bootstrap\app.php

```
<?php
$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'/../')
);
$app->bind(App\ServiceTest\ServiceContract::class,
App\ServiceTest\GeneralService::class);
$app->bind(App\ServiceTest\GeneralService::class,
App\ServiceTest\GeneralService::class);
```

文件 laravel\app\Http\routes.php

```
<?php
$generalServiceOne = $this->app->make(App\ServiceTest\
GeneralService::class);
$generalServiceTwo = $this->app[App\ServiceTest\ServiceContract::class];
```

这里的 App\ServiceTest\ServiceContract 为一个接口，而 App\ServiceTest\GeneralService 类实现了该接口，那么在绑定服务时可以通过类名或接口名作为服务名，而服务是类名。在 Laravel 框架中，这种服务为类名的会通过服务容器中的 getClosure() 函数自动生成创建该类实例对象的回调函数并进行绑定。那么对于服务名称是用类名还是用接口名好呢，答案是接口名。在 Laravel 框架中，针对不同的功能模块类设计了相应的接口，保存在 Illuminate\Contracts 目录下。使用接口绑定服务的好处就是实现松耦合设计，具体内容将会在服务解析中的依赖注入部分进行介绍。

8.1.3 服务解析

前面介绍了服务绑定的内容，当服务已经绑定到服务容器中后，就可以在之后的时间

随时获取，也可以称为服务解析。服务解析需要两个步骤，一个是获取服务容器对象，在 Laravel 框架中因为所有的功能模块都是通过服务容器实例黏合在一起，所以大部分功能类中都记录服务容器实例的属性，通常为 `$app` 属性，也可以通过 Facades 中的 `App` 外观或 `app()` 全局函数来获取；另一个是通过服务容器实现对应服务的解析。服务解析的几种实现代码如下：

文件 `laravel\app\Http\routes.php`

```
<?php
$generalServiceOne = $this->app->make(App\ServiceTest\
GeneralService::class);
$generalServiceTwo = $this->app[App\ServiceTest\GeneralService::class];
$generalServiceThree = app(App\ServiceTest\GeneralService::class);
$generalServiceFour = \App::make(App\ServiceTest\GeneralService::class);
```

上述四种方式都是通过服务名称来实现服务的解析，第一种方式是直接通过 `make()` 方法实现的；第二种是通过类似访问数组的方式解析的，因为服务容器实现了 `ArrayAccess` 接口；第三种是通过全局函数 `app()` 解析服务的，当传递参数为 `NULL` 时，则返回服务容器的实例，如果传递参数不为空，而是某个服务名称时，则会返回相应的服务；第四种是通过 Facades 中的 `App` 外观解析的。

除了上述几种服务解析方式，Laravel 框架还实现了一种更重要的方式，即依赖注入的方式。依赖在程序中随处可见，如在实例化一个类时，需要给其构造函数传递其他类实例的参数，这就是依赖。存在依赖，就需要有依赖注入，将相应的实参赋值给形参就是依赖注入，很多时候是通过手动方式实现依赖注入的，但在大型程序中这将严重增加程序员的负担，因为程序员需要管理所有依赖并手动添加，如果修改某一个类，可能需要修改很多针对该类的依赖注入。而 Laravel 框架提供了依赖自动注入，也就是说只需要关注服务容器中服务的提供，对于需要依赖的地方，服务容器会根据依赖的限制自动在容器中查找要求的服务生成实参赋值给形参，实现依赖的自动注入。实现代码如下：

文件 `laravel\app\Http\Controllers\WelcomeController.php`

```
<?php namespace App\Http\Controllers;
class WelcomeController extends Controller
{
    public $generalService;
    public function __construct(\App\ServiceTest\GeneralService
$generalService)
    {
        $this->generalService = $generalService;
    }
    public function index()
    {
```



```
dd($this->generalService);
```

这里设计了一个 `WelcomeController` 控制器类，在该类的构造函数中存在依赖 `$generalService`，而该依赖的类型是 `\App\ServiceTest\GeneralService` 类，也就是说注入的依赖必须是该类的实例，那么服务容器在创建控制器类实例的时候会检查构造函数中的依赖参数，根据依赖参数的类型查找容器中与其对应的服务名称，如果相同则会生成对应的服务并将该服务作为实参注入到控制器类的构造函数中。需要注意的是，如果在服务注册后需要使用依赖注入功能，则该服务名称和服务是需要遵守一定的规范的，即服务名称一般为服务生成的实例对象的类名称或接口名称，只有这样当服务容器根据依赖限制查找到服务后生成的实例对象才能满足这个限制，否则就会报错。

依赖自动注入并不一定只是在类的构造器中可以使用，在一些方法中也可以使用，实例代码如下：

文件：laravel/app/Http/Controllers/WelcomeController.php

```
<?php namespace App\Http\Controllers;
class WelcomeController extends Controller
{
    public function index(\App\ServiceTest\GeneralService
$generalService)
    {
        dd($generalService);
    }
}
```

通过上述实例可以看到，在 `Laravel` 框架中既可以在构造函数中实现依赖自动注入，又可以在函数中实现依赖自动注入，那是否说在 `Laravel` 框架中任何类的构造函数和方法都可以使用这种方式实现依赖注入呢？答案是否定的，通过依赖注入的方式解决依赖关系的类和方法是需要服务容器创建的类或调用的方法，也就是说通过服务容器创建的类的构造函数可以通过依赖注入的方式解决依赖问题，因为服务容器在创建该类的实例时会检查构造函数中的依赖，所以才能实现依赖自动注入，对于方法也是一样。本实例中控制器和控制器的方法都是通过服务容器自动创建和调用的，这部分内容可以参看第7章中的“路由处理生成响应”小节的内容。在本章“服务绑定”小节中介绍了尽可能通过接口进行服务绑定，而非通过具体类进行服务绑定，通过接口绑定实现依赖注入的代码如下：

文件：laravel/app/ServiceTest/ServiceContract.php

```
<?php
namespace App\ServiceTest;
interface ServiceContract
```

```
{
}
```

文件: laravel\bootstrap\app.php

```
<?php
$app = new Illuminate\Foundation\Application(realpath(__DIR__.'../../'));
$app->bind(App\ServiceTest\ServiceContract::class,
App\ServiceTest\GeneralService::class);
```

文件: laravel\app\Http\Controllers\WelcomeController.php

```
<?php namespace App\Http\Controllers;
class WelcomeController extends Controller {
    public function interfaceIndex(\App\ServiceTest\ServiceContract
$generalService)
    {
        dd($generalService);
    }
}
```

这里 \App\ServiceTest\ServiceContract 实际上是一个接口, 而 \App\ServiceTest\ GeneralService 类实现了该接口, 在服务绑定中服务名称为接口名称, 服务为具体类名称, 在服务容器中会自动创建生成回调函数的服务并实现绑定。通过这种服务绑定使得一个具体功能与一个接口实现了关联, 在以后需要应用时通过解析该接口即可获得一个具体功能类的实例, 而对于程序本身只与接口耦合, 当应用需求发生变化时可以随时修改具体类, 只要具体类符合接口规范, 那么整个程序都不需要改变依然可以正常运行, 哪怕将 Laravel 框架中所有的功能模块都换成其他的, 只要遵循接口规范, 各功能模块之间依然可以相互协作执行功能, 而 Laravel 的接口规范定义在 Illuminate\Contracts\ 文件夹下。

8.2 服务提供者

在 6.1 节“服务容器”的实例中, 服务绑定都是在 laravel\bootstrap\app.php 文件中实现的, 这种实现虽然可以完成服务绑定功能, 但是如果所有功能模块的服务都在这里绑定, 那么将会产生一个庞大而混乱的文件。在多人开发的项目中, 每个人都要修改这个文件而产生各种冲突, 显然优雅的代码是不允许存在这种情况的。Laravel 框架是通过服务提供者来解决服务绑定问题的, 在每个功能模块中都有一个服务提供者, 而服务提供者都继承了框架提供的 Illuminate\Support\ServiceProvider 抽象类, 该抽象类中提供一个虚函数 register(), 所以具体类需要实现 register() 函数, 而该函数就是用于服务绑定的。于是, 每个功能模块的服务提供者将模块中提供的服务进行绑定, 而程序在启动过程中会遍历所有的服务提供者, 并将所有服务绑定到服务容器中, 从而完成服务绑定功能, 在程序的后期处理过程中就可以

通过服务解析使用相应的功能模块了。这里依然使用 6.1 节中的实例，并使用服务提供者改造其服务绑定过程。

8.2.1 创建服务提供者

对于服务提供者类可以通过 artisan 命令 “php artisan make:provider TestServiceProvider” 来创建，而创建后 TestServiceProvider.php 文件会存放在 laravel\app\Providers\ 目录下，但有时希望像 Laravel 框架一样，某个功能模块的服务提供者放在这个功能模块的文件夹下，这时可以将生成的文件移动到相应的文件夹下。由于文件位置发生变化，所以需要修改命名空间，否则自动加载文件将会找不到。在 6.1 节中，创建了 ServiceTest 文件夹，并在该文件夹下模拟创建了功能类，可以将文件夹视为一个功能模块，于是将服务提供者移动到该文件夹下，同时将服务绑定功能从 app.php 文件移动到服务提供者的 register() 函数中。创建的服务提供者代码实现如下：

文件：laravel\app\ServiceTest\TestServiceProvider.php

```
<?php namespace App\ServiceTest;
use Illuminate\Support\ServiceProvider;
class TestServiceProvider extends ServiceProvider
{
    // 启动方法
    public function boot()
    {
    }
    // 注册应用中的服务
    public function register()
    {
        $this->app->bind(ServiceContract::class, GeneralService::class);
        $this->app->bind(GeneralService::class, GeneralService::class);
        $instance = new InstanceService();
        $this->app->instance('instanceService', $instance);
    }
}
```

在上述新创建的服务提供者中有两个方法，分别是 boot() 方法和 register() 方法，其中 register() 方法用于服务绑定，而 boot() 方法会在所有服务提供者注册完成后才被调用，这时可以在其中使用所有已经注册过的服务。虽然已经完成了服务提供者类的创建和服务绑定的函数，但框架现在还不知道多了一个服务提供者，所以在程序运行过程中还不会调用该类中的 register() 方法完成服务绑定，所以需要在某个位置进行注册来告诉框架新创建的服务提供者类。下面将介绍服务提供者的调用流程，从而注册新创建的服务提供者类。

8.2.2 注册服务提供者

在服务容器实例化 (生成全局的 `$app`) 完成后, 会注册三个服务, 分别是 Web 处理核心类、命令行处理核心类和异常类。如果接收到的是一个 Web 请求, 那么将会通过服务容器服务解析 Web 处理核心类, 得到 Web 处理核心类实例, 即全局的 `$kernel` 变量, 该实例中有一个数组属性 `$bootstrappers`, 该数组中记录了程序处理请求的准备工作需要的类, 其中就有专门处理服务提供者的 `Illuminate\Foundation\Bootstrap\RegisterProviders` 类, 然后通过 `$kernel` 类的 `handle()` 函数处理请求, 在该函数中会调用 `bootstrap()` 函数, `bootstrap()` 函数又调用服务容器实例中的 `bootstrapWith()` 函数, 而该函数将会实例化处理服务提供者的 `RegisterProviders` 类并调用该实例中的 `bootstrap()` 函数, `bootstrap()` 函数又会调用服务容器中的 `registerConfiguredProviders()` 函数, 而该函数会从配置文件中提取 Laravel 程序中所有的服务提供者类, 遍历实例化这些服务提供者并调用 `register()` 函数完成服务绑定功能。也就是说, 只要将新创建的服务提供者类在配置文件中注册即可被程序自动实例化调用, 而服务提供者的注册配置文件是 `config/app.php`, 在该文件中的配置数组有一个 `providers` 项, 该项中记录的就是所有需要注册的服务提供者。这里在该数组项中添加新创建的服务提供者, 代码如下:

```
'providers' => [
    // 省略 Laravel 框架自带的服务提供者部分代码
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,
    App\ServiceTest\TestServiceProvider::class,
],
```

8.2.3 缓载服务提供者

在前面小节的实例中, 创建的服务提供者是在程序处理请求的准备阶段进行服务绑定的, 而如果这项服务并不是所有的请求都需要, 那么就会影响应用程序的性能。好的做法是对于不是每个请求都需要使用的服务只有在需要时才临时进行服务绑定, 然后再进行服务解析。Laravel 框架为这种情况提供了缓载服务提供者, 缓载服务提供者除了具有注册方法 `register()` 外, 还需要将 `$defer` 属性设置为 `true`, 同时定义一个 `provides` 方法, 用于返回服务提供者绑定服务的名称。这里将前面小节中的服务提供者修改为缓载服务提供者, 代码如下:

文件: `laravel/app/ServiceTest/TestServiceProvider.php`

```
<?php namespace App\ServiceTest;
use Illuminate\Support\ServiceProvider;
class TestServiceProvider extends ServiceProvider
```

```

{
    protected $defer = true;
    public function boot() {
    public function register()
    {
        $this->app->bind(ServiceContract::class, GeneralService::class);
        $this->app->bind(GeneralService::class, GeneralService::class);
        $instance = new InstanceService();
        $this->app->instance('instanceService', $instance);
    }
    // 返回缓载服务提供者所绑定服务的名称
    public function provides()
    {
        return [ServiceContract::class,
                GeneralService::class,
                'instanceService'];
    }
}

```

前文已经介绍了服务提供者注册的基本过程，其中提到了服务提供者记录在配置文件 `config/app.php` 的 `providers` 项中，在注册时会遍历该数组中的所有服务提供者并判断其类型。当 `$defer` 属性为 `true` 时，说明服务提供者的类型是缓载类型，于是将 `provides()` 方法返回的服务名称数组记录下来，当通过服务容器的 `make()` 函数进行服务解析时，会判断该服务名称是否是缓载类型，如果是则调用缓载服务提供者的 `register()` 函数完成服务绑定，再进行服务解析。

本章对服务容器和服务提供者的使用进行了介绍，Laravel 框架之所以能够做到松耦合设计，与它们是分不开的，因为两者的存在实现了依赖自动注入，而依赖注入的类型又是以接口进行约束的，只要符合接口规范的模块之间都可以相互替换而不必修改框架的其他部分代码，这就是 Laravel 框架的艺术所在，这种艺术的设计理念也可以用到其他项目开发中去，构建出更多优秀的项目。

第9章

请求与响应的操作

前面介绍了请求到响应的整个过程，也知道了它们在 Laravel 框架应用程序中被封装成 `Illuminate\Http\Request` 类实例和 `Illuminate\Http\Response` 类实例，它们又都继承了 `Symfony` 框架对应的类，在整个应用程序执行过程中，它们都是被自动创建的，好像没办法对其进行操作。其实不然，在应用程序开发中，经常需要对它们进行直接的操作和处理，对于请求实例来说，更多的是通过请求实例获取想要的数据和参数，而对于响应来说可以直接生成响应实例或进行重定向，而这些内容是程序开发过程中必不可少的。

9.1 HTTP 请求实例的操作

9.1.1 请求实例的获取

要操作请求，那么首先就需要获取请求的实例，请求实例在创建成功后会在服务容器中进行备份，这部分工作其实是在调用 `Illuminate\Foundation\Http\Kernel` 类实例的 `sendRequestThroughRouter()` 方法时完成的，其实就是 “`$kernel->handle(...)`” 过程中的第二个函数调用，实现方法是 “`$this->app->instance('request', $request)`”，即在服务容器中以实例对象的形式存储。既然在服务容器中有了备份，那么在后面的程序开发中，就有三种方法可以获取到它，不过这三种方法最终都是借助于服务容器，只是途径不同而已。

方法一：直接通过外观形式访问 `Request` 类的方法，即 `facade` 方式。

```
$all = Request::all();
```

这种方法虽然无法直接获取 `Request` 类实例，但是可以任意调用 `Request` 类实例的方法。需要注意的是，这种方法需要在头部使用 “`use Request`”，这样在自动加载 `Request` 类时是直接加载 `Request` 类，而不是文件命名空间下的 `Request` 类，于是在加载过程中会通过类的外观别名加载 `Illuminate\Support\Facades\Request` 类。

方法二：直接调用服务容器获取。

```
$request = app("request");
```


这里 app() 函数是全局函数，在 Illuminate\Foundation\helpers.php 文件中定义，其实实现的就是调用服务容器实例中的 make("request") 方法得到 Request 类实例。

方法三：通过依赖注入的方法获取。

```
public function login(Request $request)
{
    $cred = $request->only('name', 'password');
}
```

方法三与方法一不同的是，需要用到的命名空间是“use Illuminate\Http\Request;”，两者通过不同的方法来调用服务容器，方法一通过外观别名，而方法三会在调用该方法时解决依赖关系，即解决 Illuminate\Http\Request 类的依赖（可以查看调试 Illuminate\Routing\ControllerDispatcher.php 文件的 resolveClassMethodDependencies() 函数来了解）。在服务容器创建中定义了核心别名，其中该类的别名就是“request”，于是这两种方法最终都是通过 make("request") 获取请求实例的。

9.1.2 请求参数的获取

对请求实例对象的应用主要是属性参数的获取和数据的存储两个方面，请求实例对象中常用的属性及对应属性中存储的内容如图 9.1 所示。

\$request	请求参数，相当于\$_POST
\$query	查询参数，相当于\$_GET
\$server	服务器及环境参数，相当于\$_SERVER
\$files	上传文件参数，相当于\$_FILES
\$cookies	请求cookie参数，相当于\$_COOKIE
\$headers	请求首部字段
\$session	session数据
\$pathInfo	请求URL
\$method	请求方法

图 9.1 请求实例对象常用属性及存储内容

可以看到，属性参数主要包括请求方法、URL、GET 数据和 POST 数据等，那么这些数据如何在程序中获取呢？下面介绍几种常用的方法并对源码进行解析。为了验证不同方法的功能，这里以 Laravel 框架中的一个注册页面为例，注册页面如图 9.2 所示。

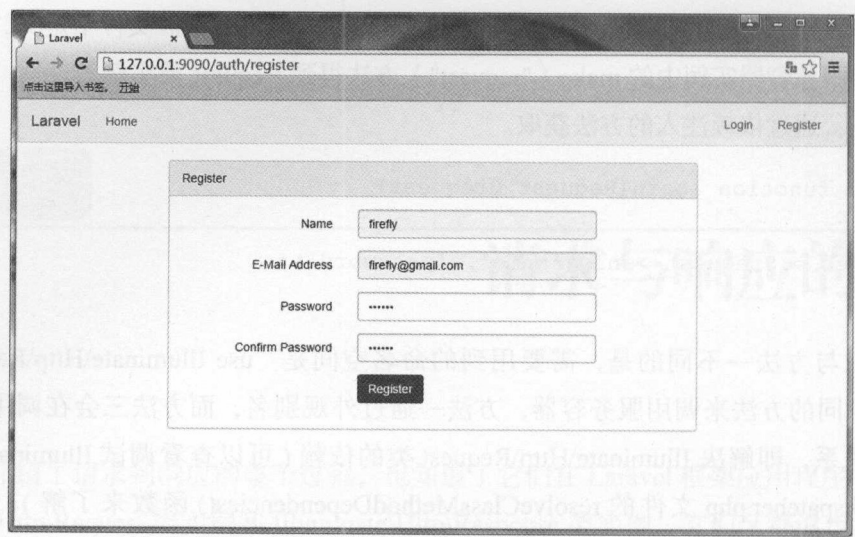


图 9.2 注册页面

将 Register 按钮的发送方法设置为“POST”，发送地址设置为“127.0.0.1:9090/auth/register?age=20”，这里加入了一个请求参数 (age=20)，该参数会被请求实例封装在内部属性中。用于测试注册请求的处理函数代码及测试结果如下：

```
public function postRegister(Request $request)
{
    $method=$request->method();
    $url=$request->url();
    $fullUrl=$request->fullUrl ();
    $uri=$request->path();
    $input = $request->input();
    $query = $request->query();
    $all = $request->all();
    $email = $request->input('email');
    $only = $request->only('email', 'password');
    $except = $request->except('email','password');
}
```

(1) 获取请求方法。

```
$method=$request->method();
```

结果：\$method ="POST"

(2) 获取请求 URL。

```
$url=$request->url();
```

结果：\$url = http://127.0.0.1:9090/auth/register

(3) 获取请求的全部 URL。

```
$fullUrl=$request->fullUrl ();
```

结果: \$fullUrl = http://127.0.0.1:9090/auth/register?age=20

(4) 获取请求 URI。

```
$uri = $request->path();
```

结果: \$uri = "auth/register"

(5) 获取所有请求输入。

```
$input = $request->input();
```

结果: 输出: \$input = array("_token"="EFp0GkxtYAeh0Qw4M6ySBns51SGxndLi45aAhk

Ib"

```
"name"="firefly"
```

```
"email"="firefly@gmail.com"
```

```
"password"="123456"
```

```
"password_confirmation"="123456"
```

```
"age"="20"
```

(6) 获取 \$_GET 中的参数。

```
$query = $request->query();
```

结果: \$query = array("age"="20")

(7) 获取所有输入数据和文件。

```
$all = $request->all();
```

结果: 这里因为没有文件输入, 所以值和 input() 方法输出的值相同。

(8) 获取请求输入。

```
$email = $request->input('email');
```

结果: \$email = "firefly@gmail.com"

(9) 获取所有输入数据中的某些数据。

```
$only = $request->only('email', 'password');
```

结果: \$only = array("email"="firefly@gmail.com", "password"="123456")

(10) 获取除某些数据以外的其他数据。

```
$except = $request->except('password');
```

结果: \$except = array ("_token" = "EFp0GkxtYAeh0Qw4M6ySBns51SGxndLi45aAhkIb", "name"="firefly", "password_confirmation"="123456", "age"="20")

通过学习上面几个利用请求实例对象获取相应请求参数的实例, 对这些方法有一个表面的认识, 下面介绍请求实例中的 all()、query() 两个方法对应的源码, 从根本上了解其实现的原理。

文件 Illuminate\Http\Request.php

```
// 从请求实例中获取所有的输入和文件
```



```

public function all()
{
    return array_replace_recursive($this->input(), $this->files->all());
}
// 检索请求实例中的输入项
public function input($key = null, $default = null)
{
    $input = $this->getInputSource()->all() + $this->query->all();
    return array_get($input, $key, $default);
}
// 从请求实例中获取输入源
protected function getInputSource()
{
    if ($this->isJson()) return $this->json();
    return $this->getMethod() == 'GET' ? $this->query : $this->request;
}
// 检索请求实例中的请求项
public function query($key = null, $default = null)
{
    return $this->retrieveItem('query', $key, $default);
}
// 从给定的源检索一个参数项
protected function retrieveItem($source, $key, $default)
{
    if (is_null($key))
    {
        return $this->$source->all();
    }
    return $this->$source->get($key, $default, true);
}

```

在上面代码的注释中，提到了请求实例的输入源，其实指的是存储请求参数的实例对象，在 7.2 节“请求实例化”部分介绍了在请求类的初始化过程中，所有请求参数都分类存储在 `ParameterBag` 类或以其为基类的实例对象中，而输入源就是指这些实例对象，其中 `ParameterBag` 类中有个方法为 `all()`，即获取该实例对象中存储的所有参数，还有一个 `get()` 方法，该方法根据输入参数的名称获取存储的参数。请求实例获取参数的方法就是基于这些实例对象和该实例对象中相应的获取方法。如在 `all()` 方法中会调用 `input()` 方法，而 `input()` 方法首先根据请求方法获取 `$query` 属性或 `$request` 属性（“`return $this->getMethod() == 'GET' ? $this->query : $this->request;`”），这两个属性实际上都是 `ParameterBag` 类的实例对象，分别存储着全局数组变量 `$_GET` 和 `$_POST` 中的值，再通过 `all()` 方法获取相应实例对象中存储的参数，实际上就是获取全局数组变量 `$_GET` 或 `$_POST` 的值。`query()` 方法就是根据输入的参数名称获取指定的请求参数，其实现原理与 `all()` 方法相同。

9.1.3 请求参数的一次存储

请求参数有时不仅来自本次请求，还需要利用上一次请求的信息，由于 HTTP 协议是无状态的，如果使用上一次请求的输入信息，就需要将其保存下来，而保存自然是以 session 或 cookie 的方式实现。Laravel 框架的请求类提供了对于请求参数的一次性存储接口，通过该接口可以将请求数据记录下来，用于处理下一次发送的请求，当下一次请求结束时删除保存的数据记录，即将请求参数存储到一次性 session 中。关于 session 的相关内容将在后面章节详细介绍，这里只介绍请求参数的一次性存储方法。用到的接口函数是请求类中的 `flash()`、`flashOnly()` 和 `flashExcept()` 三个方法。这里依然以上一节的用户注册为例：

```
$request->flash();
$request->flashOnly('email', 'password');
$request->flashExcept('password');
```

通过上述请求实例的三个方法可以实现请求参数的一次性存储，只是存储的内容有所区别，`flash()` 方法是将请求的所有参数进行存储，`flashOnly()` 方法只存储与实参对应的请求参数，`flashExcept()` 方法存储除了与实参对应的请求参数。下面给出三个函数的源码进一步了解其实现细节。

文件 `Illuminate\Http\Request.php`

```
// 将当前请求的输入存储到 session 中
public function flash($filter = null, $keys = array())
{
    $flash = ( ! is_null($filter)) ? $this->$filter($keys) : $this->input();

    $this->session()->flashInput($flash);
}

// 只存储输入中的一部分内容到 session 中
public function flashOnly($keys)
{
    $keys = is_array($keys) ? $keys : func_get_args();
    return $this->flash('only', $keys);
}

public function flashExcept($keys)
{
    $keys = is_array($keys) ? $keys : func_get_args();
    return $this->flash('except', $keys);
}
```

上面源码显示 `flash()` 函数首先获取请求的输入，即 “`$flash = (! is_null($filter)) ? $this->$filter($keys) : $this->input();`”，如果输入的第二个参数 (`$keys`) 为 `null`，则获取请求的所有输入参数，否则调用相应的方法获取过滤后的请求输入参数，然后调用 session 实例的

flashInput() 函数将输入存储到 session 中，对于 session 实例中的这个函数将会在第 12 章详细介绍，这里先了解一下就可以了。该函数会将参数存储到 session 中，并且会在下次请求结束时将内容注销。

有时候，需要重定向到其他页面，并将请求输入数据存储到一次性 session 中，这里用到的是 Redirect 类的 withInput() 方法，具体用法如下：

```
return redirect('form')->withInput();
return redirect('form')->withInput(Request::input());
```

这里 redirect('form') 返回的是 Redirect 类的实例，对于 redirect() 函数将在 9.2.3 小节中介绍。

9.1.4 获取一次存储数据

前面讲到了如何将输入数据存储到一次性 session 中，那么在下一次请求到来时如何恢复上次请求存储的数据呢？这里用到的是 Request 类中的 old() 函数。实例及 old() 函数源码如下：

```
$username = $request->old(email);
```

文件 Illuminate\Http\Request.php

// 检索旧的输入项

```
public function old($key = null, $default = null)
{
    return $this->session()->getOldInput($key, $default);
}
```

这里也是通过 session 实例读取一次性存储数据，如果想在视图的 Blade 模板中使用一次性存储数据，可以使用辅助方法 old()，例如 “{{ old('username') }}”。

通过前面几章中关于请求内容的介绍和本章中关于请求实例的操作，读者已经了解了请求的实例化过程及请求实例的结构和组成，知道在程序开发中如何调用请求实例，并使用请求实例的方法获取相关参数和操控请求实例，那么对于请求这部分内容已经了解了它的所有基本知识，如果还有其他特殊的需求，可以参看源码来学习。在了解了上面这些内容后，源码对读者来说也变得简单了。

9.2 HTTP 响应

对于请求的实例化，可以将其看做是 HTTP 请求参数封装的过程，包括请求报文的请求行、首部字段和主体三部分，而对于 HTTP 响应实例的生成，也可以看做是对响应参数的封装过程，包括响应报文的起始行、首部字段和主体三部分，最终生成的响应实例对象常用属性及存储内容如图 9.3 所示。

\$version	协议版本
\$statusCode	响应状态码
\$statusText	响应原因短语
\$headers	响应首部字段
\$content	响应主体

图 9.3 响应实例对象常用属性及存储内容

在 Laravel 框架中对于响应的生成一般有三种形式，第一种只生成响应的主体内容部分；第二种是生成响应的首部和主体部分；第三种是生成重定向的响应，即只包含响应的重定向首部。对于其他部分，Laravel 框架会根据请求来完成，如起始行其实是在响应发送时生成的，在 7.4.1 “响应的发送” 小节的 `sendHeaders()` 函数中包含语句 `“header(sprintf('HTTP/%s %s %s', $this->version, $this->statusCode, $this->statusText), true, $this->statusCode);”`，这就是生成起始行。下面将主要介绍三种生成响应的方式。

9.2.1 生成响应的主体内容

响应的主体内容可以简单地看做是一个很长的字符串，因为在发送响应的主体内容部分时，只用到语句 `“echo $this->content;”` 就实现主体内容的发送了。所以，响应主体内容的生成可以直接返回一个字符串来完成，当然也可以返回视图文件。具体实例如下：

```
public function index()
{
    return " Hello Laravel ";
}
public function index()
{
    return view('index');
}
```

第一种方法返回一个字符串作为主体内容，而第二种方法返回一个 HTML 视图文件，也相当于一个字符串。返回的主体会被 `Response` 实例保存在 `$content` 属性中，而首部会通过调用 `Illuminate\Routing\Router` 类的 `prepareResponse()` 方法生成，这部分内容在第 7 章中有介绍。

9.2.2 生成自定义响应的实例

上一小节讲述的是生成响应的主体内容，其他部分都是在 Laravel 框架中自动完成的，但是我们也可以生成自定义的响应实例，包含响应的首部字段和主体，当然也可以包含响应

起始行中版本、状态和原因短语三个参数的值。在 Laravel 框架中提供了不同方法生成响应实例，具体实例如下。

(1) 通过 new 关键字初始化新的响应实例对象，相关源码如下：

文件：laravel\app\Http\Controllers\WelcomeController.php

```
public function index()
{
    $content = "Hello WShuo";
    $status = 200;
    $type = 'text/html; charset=UTF-8';
    return (new Response($content, $status))
        ->header('Content-Type', $type);
}
```

文件：vendor\symfony\http-foundation\Symfony\Component\HttpFoundation\Response.php

// 响应类构造函数

```
public function __construct($content="", $status=200, $headers=array())
{
    $this->headers = new ResponseHeaderBag($headers);
    $this->setContent($content);
    $this->setStatusCode($status);
    $this->setProtocolVersion('1.0');
    if (!$this->headers->has('Date')) {
        $this->setDate(new \DateTime(null, new \DateTimeZone('UTC')));
    }
}
```

文件：Illuminate\Http\ResponseTrait.php

// 在响应实例中设置首部

```
public function header($key, $value, $replace = true)
{
    $this->headers->set($key, $value, $replace);
    return $this;
}
```

在使用 new 关键字生成响应实例时要包含响应类命名空间，即“use Illuminate\Http\Response;”，该类继承了 Symfony 框架的响应类，即 Symfony\Component\HttpFoundation\Response 类，在调用构造函数时调用的是 Symfony 框架中 Response 类的构造函数，而 header() 方法是在响应首部添加首部字段信息，是通过 trait 方式在 Illuminate\Http\Response 类中加入的。

(2) 通过 response() 函数的方法生成响应实例，相关源码如下：

文件：laravel\app\Http\Controllers\WelcomeController.php

```

public function index()
{
    return response()->view('welcome')->header('Content-Type','text/
html; charset=UTF-8');
}

```

文件：Illuminate\Foundation\helpers.php

```

if ( ! function_exists('response'))
{
    // 返回一个新的响应实例
    function response($content = '', $status = 200, array $headers =
array())
    {
        $factory = app('Illuminate\Contracts\Routing\ResponseFactory');
        if (func_num_args() === 0){
            return $factory;
        }
        return $factory->make($content, $status, $headers);
    }
}

```

文件：Illuminate\Routing\ResponseFactory.php

```

// 返回一个包含新视图文件的响应实例
public function view($view, $data=array(), $status=200, array $headers =
array())
{
    return static::make($this->view->make($view, $data), $status,
$headers);
}
// 返回一个新的响应实例
public function make($content="", $status=200, array $headers=array())
{
    return new Response($content, $status, $headers);
}

```

这里通过 `reponse()` 全局函数来实现响应实例的生成，如果该函数包含参数，则直接返回生成的实例，即“`return $factory->make($content, $status, $headers);`”；如果不包含参数，则返回生成响应实例的工厂，即“`return $factory;`”。在其工厂类（`ResponseFactory` 类）中通过 `view()` 方法生成包含新视图文件的响应实例，然后通过响应实例的 `header()` 方法自定义添加首部字段，这里其实在本质上是一样的，只是形式有所不同、功能有所差异而已。

9.2.3 生成重定向的响应

重定向响应实际上是 `Illuminate\Http\RedirectResponse` 类的实例，而该类继承了 `Symfony\Component\HttpFoundation\RedirectResponse` 类，这个类又继承了 `Symfony\Component\HttpFoundation\Response` 类，因此可以将重定向响应看做是一个特殊的响应，只是在响应报文首部中包含了 Location 重定向字段。Laravel 框架中的 `Illuminate\Http\RedirectResponse` 类是在 `Symfony` 框架的 `RedirectResponse` 类的基础上加入了更多的功能函数，如向 session 中存储一次性数据、自定义首部信息等。下面给出重定向响应生成的实例及部分源码：

文件：laravel\app\Http\Controllers>WelcomeController.php

```
public function index()
{
    return redirect('auth/login');
}
```

文件：Illuminate\Foundation\helpers.php

```
if ( ! function_exists('redirect'))
{
    // 获取重定向实例
    function redirect($to = null, $status = 302, $headers = array(), $secure
= null)
    {
        if (is_null($to)) return app('redirect');
        return app('redirect')->to($to, $status, $headers, $secure);
    }
}
```

文件：Illuminate\Routing\RoutingServiceProvider.php

// 注册重定向生成服务

```
protected function registerRedirector()
{
    $this->app['redirect'] = $this->app->share(function($app)
    {
        $redirector = new Redirector($app['url']);
        if (isset($app['session.store']))
        {
            $redirector->setSession($app['session.store']);
        }
        return $redirector;
    });
}
```

```
// 注册 URL 生成器服务
protected function registerUrlGenerator()
{
    $this->app['url'] = $this->app->share(function($app)
    {
        $routes = $app['router']->getRoutes();
        $app->instance('routes', $routes);
        $url = new UrlGenerator($routes, $app->rebinding(
            'request', $this->requestRebinder()
        ));
        $url->setSessionResolver(function()
        {
            return $this->app['session'];
        });
        $app->rebinding('routes', function($app, $routes)
        {
            $app['url']->setRoutes($routes);
        });
        return $url;
    });
}
```

重定向响应实例的生成过程可以分为两个步骤实现，第一步是实现重定向生成器的实例化，第二步是完成重定向响应实例的生成。第一步实际上是代码“app('redirect')”的实现过程，即通过服务容器解析名称为“redirect”的服务。“redirect”服务是在服务容器初始化时注册路由核心服务提供者(RoutingServiceProvider)的过程中注册的。Redirector 类实例中包含一个很重要的属性 \$generator，它是 URL 生成类(UrlGenerator 类)的实例，用来生成完整的请求 URL，该实例是通过 \$app('url') 由服务容器生成的，而 URL 服务也是在 RoutingServiceProvider 类中注册的。接下来将通过重定向生成器完成重定向响应实例的生成。

文件：Illuminate\Routing\Redirector.php

```
// 根据给定的路径创建重定向响应
public function to($path, $status = 302, $headers = array(), $secure =
null)
{
    $path = $this->generator->to($path, array(), $secure);
    return $this->createRedirect($path, $status, $headers);
}
// 创建一个新的重定向响应
protected function createRedirect($path, $status, $headers)
{
    $redirect = new RedirectResponse($path, $status, $headers);
    if (isset($this->session))
```

```

    {
        $redirect->setSession($this->session);
    }
    $redirect->setRequest($this->generator->getRequest());
    return $redirect;
}

```

文件: vendor\symfony\http-foundation\Symfony\Component\HttpFoundation\RedirectResponse.php

// 创建一个符合重定向状态码规则的重定向响应

```

public function __construct($url, $status = 302, $headers = array())
{
    if (empty($url)) {
        throw new \InvalidArgumentException('Cannot redirect to an
empty URL.');
```

}

```

    parent::__construct('', $status, $headers);
    $this->setTargetUrl($url);
    if (!$this->isRedirect()) {
        throw new \InvalidArgumentException(sprintf('The HTTP status
code is not a redirect ("%s" given).', $status));
    }
}

// 设置响应重定向地址
public function setTargetUrl($url)
{
    if (empty($url)) {
        throw new \InvalidArgumentException('Cannot redirect to an empty
URL.');
```

}

```

    $this->targetUrl = $url;
    $this->setContent(...);
    $this->headers->set('Location', $url);
    return $this;
}

```

在 Redirector 类实例化后, 通过调用 createRedirect() 方法实例化 Symfony 框架下的 RedirectResponse 类, 该类主要特点是包含一个 Location 首部字段, 由 setTargetUrl(\$url) 函数中的 “\$this->headers->set('Location', \$url);” 实现。

这里其实只要把握一个主要过程, 即通过 redirect() 生成一个重定向生成器, 即 Redirector 类实例, 该类完成重定向 URL 的获取及重定向实例的生成。重定向响应生成的其他形式其实都是按照这个步骤完成的, 只是实现的方法略有不同而已, 参看以下几个实例。

重定向并进行一次性数据存贮:


```
return redirect('user/login')->with('message', 'Login Failed');
```

Laravel 框架在 Symfony 框架的重定向响应类的基础上添加了一次性数据存储的函数, 可以通过 Illuminate\Http\RedirectResponse 类的实例的 with()、withInput()、onlyInput() 和 withErrors() 等函数实现。

返回前一个 URL 的重定向:

```
return redirect()->back();
```

根据路由名称及控制器的重定向:

```
return redirect()->route('login');
return redirect()->action('App\Http\Controllers\HomeController@index');
```

当 redirect() 函数不包含参数时, 实际返回的是 Redirector 类的实例, 即 “if (is_null(\$to)) return app('redirect');”, 然后调用 Redirector 类的实例的 back()、route()、action() 等方法生成请求的 URL (根据属性 \$generator, 即 UrlGenerator 类实例) 并生成重定向响应实例。

本章介绍了 Laravel 框架对 HTTP 请求和响应对象的封装和应用, 这两部分是与用户交互的手段, 因此在程序开发过程中会经常使用到。其中, 请求对象主要是处理其中的数据, 包括获取、存储等, 而响应对象主要是不同组成构建及响应的重定向等。了解了这些, 需要时再看看底层的源码实现就可以熟练掌握它们的应用了。

第 10 章

数据库及操作

对于服务器程序的设计，数据库是很重要的一部分，对应数据库接口设计的好坏决定了扩展性、开发效率和执行效率。Laravel 框架通过统一的接口实现对不同数据库操作的封装，使得对数据库的连接和操作变得非常容易，与数据库相关的配置在文件“`config/database.php`”中，可以通过修改配置文件来决定到底使用何种数据库。目前，Laravel 框架支持 MySQL、Postgres、SQLite 和 SQL Server 四种数据库。

10.1 数据库迁移与填充

数据库的迁移和填充作为开发数据库的辅助手段可以极大提高开发效率，使得对数据库的管理和控制变得简单。我们知道对于程序的源代码有版本控制工具，如 subversion 和 GIT 等，而在实际程序开发过程中，经常都是迭代的开发方式，数据库架构也会不断变化，数据库迁移实际上可以看做是数据库的版本控制，通过在 Laravel 框架下建立数据库迁移文件，可以很容易地实现不同数据结构间的切换，而数据库填充可以通过文件控制数据中的数据内容，使得程序的测试变得容易，下面将分别介绍这两部分内容。

10.1.1 数据库迁移

Laravel 的数据库迁移其实是定义了一个统一的接口来实现数据库架构的创建和维护，而这种统一的接口与底层的数据库及其操作语言都是无关的，Laravel 中通过 PHP 语言来定义这些接口并实现数据库架构的描述，当需要将这个数据库架构移植到所支持的数据库后端时，只需要执行这个描述文件就可以了，这样就为数据库架构的修改和维护提供了极大的方便。

一个数据库迁移文件对应的是在“`laravel/database/migrations/`”目录下的一个 PHP 文件，迁移文件使用 PHP 语言来描述，即使对 SQL 语句不了解，一样可以实现数据库架构的设计，而文件的执行是通过 `artisan` 命令来完成的，需要注意的是，`artisan` 命令需要在 Laravel 框架的根目录下执行。接下来具体介绍数据库迁移的相关步骤。

(1) 数据库配置。

要实现数据库迁移，首先需要对数据库账号信息进行配置，而对数据库账号信息的配置文件有两个，分别是“laravel\config\database.php”和“laravel\.env”，其中 .env 文件是主配置文件，因为该文件中配置的参数会覆盖 database.php 中的参数，这里主要修改 .env 中的配置参数，主要有数据库位置、数据库名、账号名和密码，这些参数根据自己的计算机环境进行配置即可。

(2) 迁移文件的建立。

完成配置后，就可以创建迁移文件了，迁移文件需要存储在“laravel\database\migrations\”目录下，而文件的名字则需要以 YYYY_MM_DD_HHMMSS_create_tableName_table.php 形式创建，其中前面是一个 UTC 时间戳识别，后面 tableName 为数据库表的名称，当然也可以直接使用 artisan 命令创建迁移文件，命令为“php artisan make:migration create_tableName_table”。通过 artisan 命令创建的文件已经在文件中给出了迁移文件的基本结构，其中包含两个方法，分别是 up() 方法和 down() 方法，其中 up() 方法是执行迁移命令时创建的表结构，而 down() 方式是执行回滚时删除的表结构。

(3) 迁移文件的设计。

数据库迁移文件的设计实际上就是对数据表的操作和维护，在 Laravel 框架中是通过结构生成器（Schema 类）来实现的，该类提供了一组与数据库无关的数据表生成方法，通过这些方法可以实现数据表的创建、删除和修改等功能。下面是一个简单的迁移文件实例。

文件：laravel\database\migrations\2014_10_12_000000_create_users_table.php

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateUsersTable extends Migration
{
    // 执行迁移
    public function up()
    {
        Schema::create('users',function(Blueprint $table){
            $table->increments('id');
            $table->string('username',32);
            $table->string('account',32) ->unique();
            $table->string('password',60);
            $table->rememberToken();
            $table->unsignedInteger('addtime');
            $table->tinyInteger('state')->unsigned()->default(1);
        });
    }
    // 回滚迁移
    public function down()
```



```
{
    Schema::drop('users');
}
}
```

这里通过 artisan 命令创建了迁移文件，只需要在 up() 方法和 down() 方法中添加相应的结构生成器方法来实现对数据表的操作，上面的实例在 up() 方法中通过 Schema 类 create() 方法创建一个名为 “users” 的数据表，该数据表中包含一个自动增量的 id、32 位长度的名字和账号、60 位长度的密码、记录令牌（rememberToken，相当于 100 位长度 varchar）、无符号整型的时间和小整型的状态，而在数据表生成器中还有更多的字段类型、字段修饰和创建索引方法可以使用，这些方法及与数据库对应的描述如表 10.1、表 10.2 和表 10.3 所示。

表 10.1 字段类型方法表

添加字段类型函数	功能描述
\$table->bigIncrements('id');	id 自动增量，使用相当于 bigInteger 类型
\$table->bigInteger('votes');	相当于 bigInt 类型
\$table->binary('data');	相当于 blob 类型
\$table->boolean('confirmed');	相当于 boolean 类型
\$table->char('name', 4);	相当于 char 类型，并带有长度
\$table->date('created_at');	相当于 date 类型
\$table->dateTime('created_at');	相当于 dateTime 类型
\$table->decimal('amount', 5, 2);	相当于 decimal 类型，并带有精度与基数
\$table->double('column', 15, 8);	相当于 double 类型，总共有 15 位数，在小数点后面有 8 位数
\$table->enum('choices', array('foo', 'bar'));	相当于 enum 类型
\$table->float('amount');	相当于 float 类型
\$table->increments('id');	相当于 Incrementing 类型（数据表主键）
\$table->integer('votes');	相当于 integer 类型
\$table->json('options');	相当于 json 类型
\$table->longText('description');	相当于 longText 类型
\$table->mediumInteger('numbers');	相当于 mediumInt 类型
\$table->mediumText('description');	相当于 mediumText 类型
\$table->morphs('taggable');	加入整数 taggable_id 与字串 taggable_type
\$table->nullableTimestamps();	与 timestamps() 相同，但允许 NULL
\$table->smallInteger('votes');	相当于 smallInt 类型

续表

添加字段类型函数	功能描述
<code>\$table->tinyInteger('numbers');</code>	相当于 <code>tinyInt</code> 类型
<code>\$table->softDeletes();</code>	加入 <code>deleted_at</code> 字段于软删除使用
<code>\$table->string('email');</code>	相当于 <code>varchar</code> 类型
<code>\$table->string('name', 100);</code>	相当于 <code>varchar</code> 类型，并指定长度
<code>\$table->text('description');</code>	相当于 <code>text</code> 类型
<code>\$table->time('sunrise');</code>	相当于 <code>time</code> 类型
<code>\$table->timestamp('added_on');</code>	相当于 <code>timestamp</code> 类型
<code>\$table->timestamps();</code>	加入 <code>created_at</code> 和 <code>updated_at</code> 字段
<code>\$table->rememberToken();</code>	加入 <code>remember_token</code> 字段，使用类型为 <code>varchar(100)</code>

表 10.2 修饰方法表

添加字段修饰	功能描述
<code>\$table->float('amount')->first();</code>	将此字段放在表的首位 (只能在 MySQL 中使用)
<code>\$table->float('amount')->after('column')</code>	将此字段方法放在其他字段后面 (只能在 MySQL 中使用)
<code>\$table->string('email')->nullable();</code>	表示此字段允许 <code>NULL</code>
<code>\$table->float('amount')->default(\$value)</code>	指定此字段的默认值
<code>\$table->integer('votes')->unsigned();</code>	配置整数为非负数

表 10.3 索引方法表

索引类型	功能描述
<code>\$table->primary('id');</code>	加入主键 (primary key)
<code>\$table->primary(array('first', 'last'));</code>	加入复合键 (composite keys)
<code>\$table->unique('email');</code>	加入唯一索引 (unique index)
<code>\$table->index('state');</code>	加入基本索引 (index)

数据表字段生成时是支持链式操作的，如实例中关于账户和状态字段的设置分别是 “`$table->string('account',32) ->unique();`” 和 “`$table->tinyInteger('state')->unsigned()->default(1);`”，其中账户 (account) 字段设置为 32 位 char 类型，其实 string 函数相当于可以设置长度的 varchar(变长度字符型) 类型，而状态 (state) 被设置为无符号的小整型，并通过 default(1) 设置默认值为 1。

(4) 执行数据迁移。

执行数据迁移可以通过 artisan 命令来完成，针对上述实例，执行迁移命令实现 “users” 数据表的建立，而执行回滚迁移实现 “users” 数据表的删除。具体命令如下：

```
php artisan migrate      执行迁移
php artisan migrate:rollback  执行回滚迁移
```

通过上面四个步骤，就可以完成一次数据库迁移，对于复杂的数据库架构，可以在一个文件中设计多个数据表，也可以在多个文件中设计数据表，进而实现对数据库版本的控制。

10.1.2 数据库填充

在对程序进行测试时，很多时候需要数据库有相应的测试数据，如果这些测试数据需要手动通过 SQL 语句来加入，那将会是件非常麻烦的事情，好在 Laravel 框架提供了数据库填充功能，可以通过 PHP 程序设计数据库的填充数据，并通过 artisan 命令自动执行填充，这样就可以实现对数据库中数据的版本控制。下面介绍数据库填充的相关步骤。

(1) 构建模型类。

模型类在 Laravel 数据库的操作中是个很重要的概念，可以将一个模型类看做是一个封装数据表的类，通过模型类就可以对这个数据表进行相应的操作。模型类将会在 Eloquent ORM 中详细介绍。下面给出针对“users”数据表的模型类代码：

```
文件 laravel\app\User.php

<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
    public $timestamps = false;
}
```

可以看到模型类非常简单，简单到无法看到这个类到底操作哪个数据表，其实对于模型类是可以设置数据表名称这个属性的，如果不设置，则默认以类名首字母小写后加上“s”作为数据表名称，如上例中定义类名为“User”，所以数据表名称就为“users”，就是在数据库迁移过程中创建的数据表。将 \$timestamps 属性设置为“false”，是因为默认情况下，对数据库添加数据时会自动添加 Laravel 框架定义的两个时间字段，而在数据库迁移时定义的时间字段不是 Laravel 框架提供的时间字段，所以这里将其设置为“false”，表示在添加数据时不再添加这两个时间字段，因为数据表中就没有设置 Laravel 定义的这两个时间字段。有了数据表名，那么对模型类的任何操作就会直接操作数据表，包括数据的增加、删除、修改和查找。

(2) 数据库填充文件的设计。

有了数据表的模型类，就可以操作数据表了，也就可以进行数据库填充文件的设计了。数据库填充文件一般放置在“laravel\database\seeds\”目录下，该文件以一个类的形式创建，类名可以任意取，不过一般约定类名为数据表名后加上“TableSeeder”，对于这个实例，

类名为“UserTableSeeder”。需要注意的是，文件名要与类名相同，否则无法自动加载类，同时该类要继承自“Illuminate\Database\Seeder”类，也可以通过 artisan 命令“php artisan make:seeder UserTableSeeder”自动创建填充文件。数据库填充设计代码如下：

文件 laravel/database/seeds/UserTableSeeder.php

```
<?php
use App\User;
use Illuminate\Database\Seeder;
class UserTableSeeder extends Seeder
{
    public function run()
    {
        DB::table("users")->delete();
        for($i=0; $i<10; $i++){
            User::create([
                'username'=>'username',
                'account'=>'account'.$i,
                'password'=>'password'.$i,
                'addtime'=>time(),
                'state'=>1
            ]);
        }
    }
}
```

这里首先通过“DB::table("users")->delete();”语句将数据表中的数据删除，然后利用循环，通过“User”模型类的 create() 方法完成数据表中数据的添加，create() 方法需要一个关联数组作为参数，而这个数组就是数据表中需要添加的数据，键为数据表字段名称，值为数据表中对应字段需要添加的数据。这些方法的实现细节在后续章节中会详细介绍，这里只要先清楚每一步的作用即可。

完成上面填充文件的设计后，还需要做一件工作，即在相应的位置调用填充文件，这个位置就是 Laravel 框架中自带的填充文件“DatabaseSeeder.php”，也就是说在执行填充命令时，调用的是该类中的 run() 方法，而在该方法中注册填充文件，就会调用到自己设计的填充方法，从而完成数据填充。具体代码如下：

文件 laravel/database/seeds/DatabaseSeeder.php

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        Model::unguard();
```

```
$this->call(UserTableSeeder::class);
Model::reguard();
}
}
```

(3) 执行数据库填充。

执行数据迁移可以通过 artisan 命令“php artisan db:seed”来完成数据填充。执行完填充后，通过 phpMyAdmin 查看填充后的结果，如图 10.1 所示。



图 10.1 数据填充结果

10.2 查询构造器

在 Laravel 中，对数据库的访问可以通过查询构造器或 Eloquent ORM 实现，不同的方法实现不同的底层封装，这两种封装都有同一个目的，即对底层不同数据库提供统一的操作接口。默认情况下，Laravel 支持四种数据库系统，即 MySQL、Postgres、SQLite 和 SQL Server。也就是说，当服务器程序数据库改变时（如从 SQLite 迁移到 MySQL 上），如果对数据库的操作是通过查询构造器或 Eloquent ORM 实现的，那么只需要修改数据库配置文件的相关配置，对于程序不需要进行任何修改就可以实现，这样将会极大提高 Laravel 框架的扩展性。在 Laravel 的官方文档中，只是提供了这两种实现的方法，而对于底层的实现方式和原理没有说明，程序开发者需要真正地了解底层的原理和实现才能在学习触类旁通，在开发时得心应手。本节先介绍查询构造器的实现原理，Eloquent ORM 的实现原理将在下节中介绍。

查询构造器的底层其实用到的还是 PHP 数据库抽象层的 PDO 扩展，该扩展是一个“轻量级”的数据库扩展，数据库的操作需要自己设计 SQL 语句来执行，而 Laravel 框架的查询构造器是在 PDO 扩展基础上设计的一个“重量级”的数据库扩展，它将 SQL 语句的设计都

进行了封装，将需要变化的部分进行了分离并以配置文件的方式进行设置，对数据库常用的操作进行了封装，提供了统一的接口，更加方便使用。本节将通过 PHP 中数据库的操作、数据库连接的封装、查询构造器的实现、查询构造器的使用和查询构造器的数据库操作五个部分进行介绍，介绍过程将以 MySQL 数据库为例。

10.2.1 PHP 中数据库的操作

一般关系型数据库采用的是“客户机/服务器”的体系结构，客户端通过 SQL 语句操作服务端。在 PHP 中，有两大类操作数据库的扩展，一种是针对各数据库开发的专用扩展，如 MySQL 扩展、SQLite3 扩展；另一种是数据库抽象层，如 PDO（PHP Data Objects，PHP 数据对象）、ODBC（Open Database Connectivity，开放数据库互连）。在 PHP 开发中，以前的开发中大多使用专用扩展，因为 PDO 是在 PHP5.1 版本才引入的，而目前开发中大多使用 PDO 扩展。Laravel 框架中关于数据库操作的底层使用的就是 PDO 扩展。

那么这两种扩展有什么区别呢？一是开发思想不同，针对各数据库开发的专用扩展采用的是面向过程的思想开发的，而 PDO 扩展采用的是面向对象的思想开发的。二是提供的接口针对的数据库范围不同，专用扩展只是针对某一种数据库开发的，如 MySQL 扩展提供的接口只能操作 MySQL 数据库，当更改底层数据库时，就要重新安装相应数据库的扩展，也需要对数据库的操作代码重新编程；而数据库抽象层则针对多种数据库提供统一的接口，采用这种接口编写的程序，在更改底层数据库时，不需要重新安装数据库抽象层，但需要安装对应的数据库驱动，代码部分只需要修改少许的连接和设置等操作，其他对数据的操作不需要修改，扩展性好，其作用方式如图 10.2 和图 10.3 所示。三是安全性，在关系型数据库中，使用专用扩展需要解决 SQL 注入的问题，而使用 PDO 扩展可以避免这个问题。

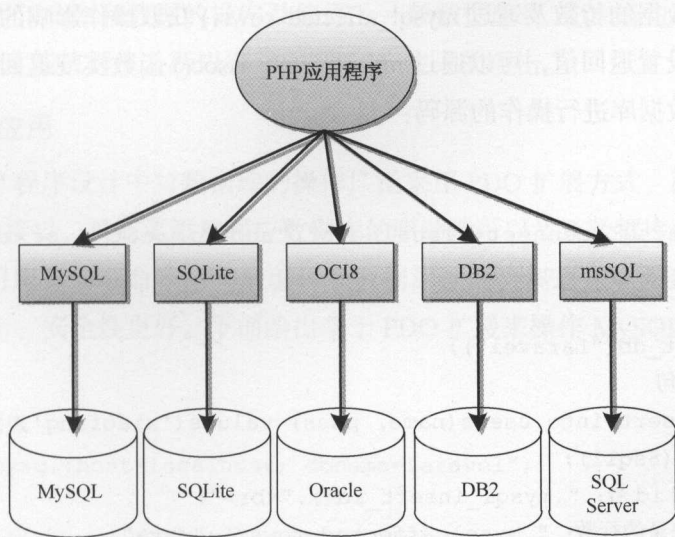


图 10.2 通过专用数据库扩展操作数据库方式

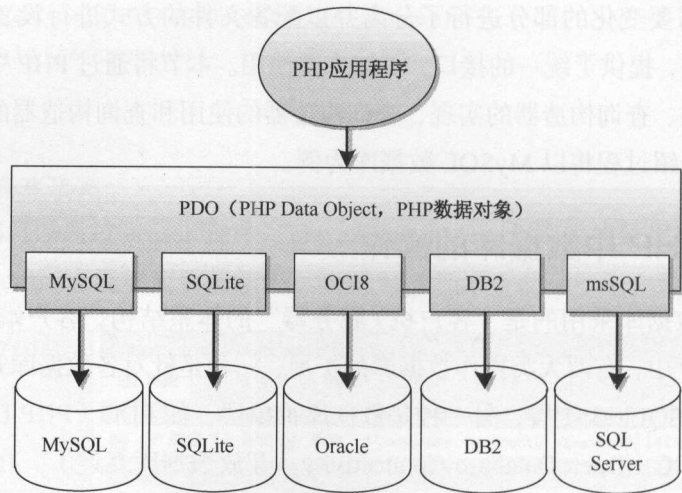


图 10.3 PDO 扩展操作数据库方式

下面针对 MySQL 数据库介绍专用扩展和 PDO 扩展的用法，一是使读者了解两者的区别和效果，二是有助于读者了解 Laravel 框架关于数据库操作底层的实现，这部分内容将会在后面章节详细介绍。

1. MySQL 扩展应用

前面提到，专用 MySQL 扩展是采用面向过程的思想开发的，当 PHP 安装了 MySQL 扩展后，就可以直接使用相应的函数来执行 SQL 语句，从而完成对数据库的操作，这里用到的是 `mysql_query()` 函数执行 SQL 语句。一般将 SQL 语句分为两类：一类是执行 SQL 语句后有返回结果的，如 `select` 语句、`desc` 语句等；第二类是没有返回结果的，如 `delete` 语句、`insert` 语句等。对于没有返回结果的语句不需要设置返回值，可以通过 `mysql_insert_id()` 函数获取最后操作数据的行数及通过 `mysql_affected_rows()` 获取操作影响的行数。对于有返回结果的语句需要设置返回值，可以通过 `mysql_fetch_assoc()` 函数实现返回值的获取。下面给出一个简单的对数据库进行操作的源码：

```
<?php
// 连接数据库
$link = mysql_connect("localhost","root","root") or die( "Could not
connect: " . mysql_error ());
// 选择数据库
mysql_select_db("Laravel");
// 执行插入语句
$sql1 = "insert into users(name, pass) values('xiaofang', '111111')";
mysql_query($sql1);
echo " 插入的 id 号: ".mysql_insert_id()."<br>";
echo " 影响记录的行数: ".mysql_affected_rows()."<br>";
$sql2 = "insert into users(name, pass) values('shuoshuo','222222')";
mysql_query($sql2);
```

```

echo " 插入的 id 号: ".mysql_insert_id()."<br>";
echo " 影响记录的行数: ".mysql_affected_rows()."<br>";
// 执行查询语句
$sql3 = "select * from users";
$result = mysql_query($sql3);
while($row = mysql_fetch_assoc($result)){
    print_r($row);
    echo "<br>";
}
// 执行修改语句
$sql4 = "update users set name='shuoshuo' where id=3";
mysql_query($sql4);
echo " 更新影响记录的行数: ".mysql_affected_rows()."<br>";
// 关闭数据库
mysql_close($link);
输出结果:
插入的 id 号: 1
影响记录的行数: 1
插入的 id 号: 2
影响记录的行数: 1
Array ( [id] => 1 [name] => shuoshuo [pass] => 111111 )
Array ( [id] => 2 [name] => shuoshuo [pass] => 222222 )
更新影响记录的行数: 1

```

通过上述源码可以看出，操作一个数据库需要四个步骤：一是连接数据库的服务端；二是选择数据库；三是准备 SQL 语句并执行；四是关闭数据库。这些步骤所用到的函数都是对应数据库专用的扩展函数，在扩展方面无法迁移到其他数据库，在安全性方面，因为是一次性完成 SQL 语句命令和数据的设定及编译（一般数据库将 SQL 语句从客户端发送到服务端时需要先编译才能执行），所以容易受到 SQL 注入的攻击。

2. PDO 扩展应用

目前，在 PHP 程序设计中数据库的操作广泛采用 PDO 扩展方式，因为该扩展对不同数据库提供统一的接口，只需要添加相应数据库的驱动就可以实现数据库的迁移，扩展性更好。同时，PDO 将服务器端命令语句的编译和数据添加的分离思想应用到客户端，可以避免 SQL 注入的攻击，安全性更好。下面给出基于 PDO 扩展来操作 MySQL 数据库的源码：

```

<?php
// 连接数据库
$dsn = "mysql:host=localhost; dbname=Laravel";
try{
    $pdo = new PDO($dsn, "root", "root");
}catch(PDOException $e){
    echo " 连接失败 ".$e->getMessage()."<br>";
}

```

```

}

// 使用 PDO 类中的 exec() 方法插入数据
$sql1 = "insert into pdousers(name, pass) values('xiaofang','111111')";
$row = $pdo->exec($sql1);
echo " 插入语句影响的行号为: ".$pdo->lastInsertId()."<br>";

// 使用 PDOStatement 类中的 execute() 方法插入数据
$sql2 = "insert into pdousers(name, pass) values(:name,:pass)";
$sta = $pdo->prepare($sql2);
$sta->execute(array("name"=>'shuoshuo',"pass"=>'222222'));
echo " 插入语句影响的行号为: ".$pdo->lastInsertId()."<br>";
$sta->execute(array("name"=>"shuoshuo2","pass"=>"212121"));
echo " 插入语句影响的行号为: ".$pdo->lastInsertId()."<br>";

// 通过 PDOStatement 的 execute() 方法执行查询语句
$sql5 = "select * from pdousers where id>:id";
$pdoSta = $pdo->prepare($sql5);
$pdoSta->execute(array("id"=>0));
while($row = $pdoSta->fetch(PDO::FETCH_ASSOC)){
    print_r($row);
    echo "<br>";
}

```

输出结果:

插入语句影响的行号为: 1

插入语句影响的行号为: 2

插入语句影响的行号为: 3

```

Array ( [id] => 1 [name] => xiaofang [pass] => 111111 )
Array ( [id] => 2 [name] => shuoshuo [pass] => 222222 )
Array ( [id] => 3 [name] => shuoshuo2 [pass] => 212121 )

```

通过上述源码可以看出，采用 PDO 扩展对数据库操作的步骤与采用专用 MySQL 扩展对数据库操作的步骤几乎相同，但在实现方式上却不同。一是在 PDO 扩展中对 MySQL 数据库的操作方法中几乎没有与该数据库关联的部分，只有在建立连接时构造 DSN（Data Source Name，数据源名称）时以“mysql:host=localhost; dbname=Laravel”的方式声明了数据库类型，而具体对数据库的操作方法与数据库类型无关，所以在数据库迁移时，只需要修改这个连接参数就可以实现，扩展性更好；二是在执行数据库操作时，可以先通过 PDO 类的 prepare() 方法将操作命令发送到服务端编译，再通过 PDOStatement 类的 execute() 方法将数据发送到服务端并执行，这种方法既可以提高多次执行同一条操作命令时的速度，也可以避免 SQL 注入的危险，所以效率、安全性都更高。

10.2.2 数据库连接的封装

在 PDO 扩展中, 可以看到虽然提供了对数据库操作的统一接口函数, 但是在与数据库连接时还是与数据库的类型存在关联, 在数据库迁移时同样需要修改这部分内容。这里所说的数据库迁移与第一节中介绍的数据库迁移与填充中的迁移有所区别, 第一节中所说的数据库迁移是指通过 PHP 文件的形式创建数据库表结构, 可以很容易实现数据库升级及数据表从一种数据库迁移到另一种数据库中, 而这里所说的数据库迁移是指应用程序从一种数据库迁移到另一种数据库上。

在 Laravel 框架中, 将这部分需要变化的部分进行了分离, 以配置文件的方式给出, 只需要修改配置文件就可以实现对不同数据库的连接和操作, 这也体现了设计模式中的封装变化原则, 即找出应用中可能需要变化之处, 把它们独立出来, 不要和那些不需要变化的代码混在一起 (Head First 设计模式)。当然, 对于不同数据库还有很多其他不同的地方 (如语法构造的不同、结果处理的不同), Laravel 框架也将这些针对不同数据库的接口封装成了类, 根据配置文件的不同, 在调用统一接口方法时, 底层其实是使用不同的类方法来实现。目前, Laravel 框架支持四种关系型数据库, 即 MySQL、Postgres、SQLite 和 SQL Server。所以只有了解了底层实现的方式, 在开发过程中才能得心应手, 如果需要添加对其他数据库的支持, 也可以正确修改需要变化的部分。下面给出数据库连接创建过程的源码, 由于这部分代码较多, 所以只给出关键部分, 并通过几个阶段进行分析。

1. 数据库管理器阶段

在服务注册阶段, 通过服务提供者 “Illuminate\Database\DatabaseServiceProvider” 注册了数据库管理器服务 (“DB” 服务) 和数据库连接工厂服务 (“db.factory” 服务), 通过上述服务可以获取数据库管理器实例 (DatabaseManager 类实例) 和数据库连接工厂实例 (ConnectionFactory 类实例), 其中数据库连接工厂实例作为数据库管理器实例的一个属性。下面介绍查询构造器实现过程中的数据库管理器阶段, 其中部分实现代码如下:

```
文件 \app\Http\Controllers\WelcomeController.php
// 使用查询构造器获取数据信息
public function index()
{
    $users = DB::table('users')->get();
}
```

对于查询构造器, 当需要操作数据库时, 可以使用类似 “DB::table('users')->get();” 的语法, 其中 “DB::table(‘users’)” 部分是获取查询构造器, 而 “->get()” 是调用查询构造器的方法实现相应数据表操作的过程。查询构造器的建立过程可以分为两个阶段: 一个是数据库连接封装阶段, 另一个是查询构造器生成阶段。本小节主要讨论数据库连接封装阶段, 而这部分又可以分为四个步骤: 一是数据库管理器阶段, 二是数据库连接工厂阶段, 三是数据库

连接器阶段，四是数据库连接创建阶段。数据库管理器阶段主要用于生成数据库连接工厂，而数据库连接工厂管理着不同数据库连接器的创建，这个阶段会根据数据库配置信息生成相应数据库的连接器，在数据库连接器阶段会根据数据库配置信息生成相应数据库的 PDO 实例，而在数据库连接创建阶段则生成相应数据库的连接实例，用于封装对应的 PDO 实例、数据库配置等信息。下面介绍数据库管理器阶段。

文件 \Illuminate\Database\DatabaseManager.php

```
// 动态地将方法传递给默认连接，此时 $method = "table"
public function __call($method, $parameters)
{
    return call_user_func_array([$this->connection(), $method],
    $parameters);
}
```

对于“DB:”阶段，是通过 DB 的外观类 (Facades) 获取数据库管理器实例并调用其中的 table() 方法，因为该类中没有这个方法，所以调用魔术函数 __call() 方法，该方法进而调用方法 “[\$this->connection(), \$method]”，即 \$this->connection() 返回实例的 table() 方法，而 “\$this->connection()” 将根据配置文件获取数据库连接实例，这也是数据库连接封装阶段的重点。

文件 \Illuminate\Database\DatabaseManager.php

```
// 获取数据库连接实例
public function connection($name = null)
{
    list($name, $type) = $this->parseConnectionName($name);
    if (!isset($this->connections[$name])) {
        $connection = $this->makeConnection($name);
        $this->setPdoForType($connection, $type);
        $this->connections[$name] = $this->prepare($connection);
    }
    return $this->connections[$name];
}

// 将连接的名称和读 / 写类型进行解析得到数据，此时 $name = null
protected function parseConnectionName($name)
{
    $name = $name ?: $this->getDefaultConnection();
    return Str::endsWith($name, [':read', ':write'])
        ? explode(':', $name, 2) : [$name, null];
}

// 获取默认连接的名字
public function getDefaultConnection()
{
}
```

```

        return $this->app['config']['database.default'];
    }
    // 创建数据库连接实例，此时 $name="mysql"
    protected function makeConnection($name)
    {
        // 获取关于该名字数据库的配置
        $config = $this->getConfig($name);
        if (isset($this->extensions[$name])) {
            return call_user_func($this->extensions[$name], $config, $name);
        }
        $driver = $config['driver'];
        if (isset($this->extensions[$driver])) {
            return call_user_func($this->extensions[$driver], $config, $name);
        }
        return $this->factory->make($config, $name);
    }

```

首先，通过 `parseConnectionName()` 函数获取配置文件中关于默认数据库名称和类型的设置，而对于数据库配置信息的获取是通过 `getDefaultConnection()` 函数实现的，其中，配置文件为“`laravel\config\database.php`”，对数据库名称和类型的配置项为“`default`”项，Laravel 框架默认设置为“`mysql`”，于是返回的数据库名称和类型配置信息分别为“`$name="mysql"`”和“`$type="null"`”。

其次，通过 `getConfig()` 函数根据数据库名称获取对数据库的配置，包括数据库的驱动、主机地址、用户名、密码和使用的数据库名等，对于不同类型的数据库配置也不同。接下来进入数据库连接工厂阶段，需要注意的是，数据库连接工厂的实例（“`db.factory`”服务）在创建数据库管理器（“`DB`”服务）的时候作为依赖已经通过服务容器注入到数据库管理器中了。

2. 数据库连接工厂阶段

前面提到 Laravel 框架默认支持四种关系型数据库，那么为了对上层提供统一的接口，需要在底层根据不同的配置调用不同的数据库驱动扩展，而这里 Laravel 框架使用了简单工厂设计模式，用来根据配置文件获取不同的数据库连接实例。下面给出数据库连接工厂阶段部分代码：

文件 `\Illuminate\Database\Connectors\ConnectionFactory.php`

```

// 基于配置创建一个 PDO 连接
public function make(array $config, $name = null)
{
    $config = $this->parseConfig($config, $name);
    if (isset($config['read'])) {
        return $this->createReadWriteConnection($config);
    }
}

```



```

        return $this->createSingleConnection($config);
    }
    // 创建一个单独的连接实例
    protected function createSingleConnection(array $config)
    {
        $pdo = $this->createConnector($config)->connect($config);
        return $this->createConnection($config['driver'], $pdo,
            $config['database'], $config['prefix'], $config);
    }
    // 基于配置创建一个连接器实例
    public function createConnector(array $config)
    {
        if (!isset($config['driver'])) {
            throw new InvalidArgumentException('A driver must be
specified.');
```

```

        }
        if ($this->container->bound($key = "db.connector.
{$config['driver']}")) {
            return $this->container->make($key);
        }
        switch ($config['driver']) {
            case 'mysql':
                return new MySqlConnection;
            case 'pgsql':
                return new PostgresConnector;
            case 'sqlite':
                return new SQLiteConnector;
            case 'sqlsrv':
                return new SqlServerConnector;
        }
        throw new InvalidArgumentException("Unsupported driver
[{$config['driver']}]");
    }
}

```

上述代码根据数据库名称实例化对应的连接器，即 `createConnector()` 函数部分。连接器实例用来管理数据库连接创建阶段 DSN 字符串的产生、PDO 类的实例化及数据库配置的设置等。在 Laravel 框架默认配置中，使用 MySQL 数据库，即读写类型用同一个数据库，如果需要设计读写分离的数据库，可以在配置时对类型进行定义，之后将会在这部分起作用。

3. 连接器阶段

在连接器阶段，针对不同的数据库将会有不同的实现，主要包括连接 DSN 名称及配置等。Laravel 框架用四个类分别封装了默认支持的四个数据库连接的过程，通过 `connect()` 方法提供统一的接口。下面给出这部分实现的代码：

文件 \Illuminate\Database\Connectors\MySQLConnector.php

// 创建一个数据库连接

```
public function connect(array $config)
{
    $dsn = $this->getDsn($config);
    $options = $this->getOptions($config);
    $connection = $this->createConnection($dsn, $config, $options);
    if (isset($config['unix_socket'])) {
        $connection->exec("use `{ $config['database'] }`;");
    }
    $collation = $config['collation'];
    $charset = $config['charset'];

    $names = "set names '$charset'".
        (!is_null($collation) ? " collate '$collation'" : '');
    $connection->prepare($names)->execute();
    if (isset($config['timezone'])) {
        $connection->prepare(
            'set time_zone="' . $config['timezone'] . '"'
        )->execute();
    }
    if (isset($config['strict']) && $config['strict']) {
        $connection->prepare("set session sql_mode='STRICT_ALL_
TABLES'")->execute();
    }
    return $connection;
}

// 根据配置创建一个 DSN 字符串
protected function getDsn(array $config)
{
    return $this->configHasSocket($config) ? $this->getSocketDsn($config)
: $this->getHostDsn($config);
}

protected function getHostDsn(array $config)
{
    extract($config);
    return isset($port)
        ? "mysql:host={$host};port={$port};dbname={$database}"
        : "mysql:host={$host};dbname={$database}";
}
```

文件 \Illuminate\Database\Connectors\Connector.php

// 创建一个新的 PDO 连接

```
public function createConnection($dsn, array $config, array $options)
```

```

{
    $username = Arr::get($config, 'username');
    $password = Arr::get($config, 'password');
    return new PDO($dsn, $username, $password, $options);
}

```

从源码中可以看到，这里使用 PDO 扩展来实现连接的建立，即“new PDO(\$dsn, \$username, \$password, \$options)”，而对于不同的 DSN 名称，则通过 getDsn() 函数获取。在连接器阶段主要完成 PDO 类的实例化及对字符集、时区等的设置，即代码“\$connection->prepare(\$names)->execute();”和“\$connection->prepare('set time_zone= "'. \$config['timezone']. '")->execute();”部分。

4. 数据库连接创建阶段

不同数据库连接的实例其实是封装了对应连接的 PDO 类实例、请求语法类实例和结果处理类实例，从而为上层使用数据库连接实例提供统一的接口。部分代码如下：

文件 \Illuminate\Database\Connectors\ConnectionFactory.php

```

// 创建一个新的连接实例
protected function createConnection($driver, PDO $connection, $database,
$prefix = '', array $config = [])
{
    if ($this->container->bound($key = "db.connection.{$driver}")) {
        return $this->container->make($key, [$connection, $database,
$prefix, $config]);
    }
    switch ($driver) {
        case 'mysql':
            return new MySqlConnection($connection, $database, $prefix,
$config);
        case 'pgsql':
            return new PostgresConnection($connection, $database, $prefix,
$config);
        case 'sqlite':
            return new SQLiteConnection($connection, $database, $prefix,
$config);
        case 'sqlsrv':
            return new SqlServerConnection($connection, $database,
$prefix, $config);
    }
    throw new InvalidArgumentException("Unsupported driver [$driver]");
}

```

文件 \Illuminate\Database\Connection.php


```

    public function __construct(PDO $pdo, $database = '', $tablePrefix = '',
        array $config = [])
    {
        $this->pdo = $pdo;
        $this->database = $database;
        $this->tablePrefix = $tablePrefix;
        $this->config = $config;
        $this->useDefaultQueryGrammar();
        $this->useDefaultPostProcessor();
    }
    // 设置默认实现的请求语法
    public function useDefaultQueryGrammar()
    {
        $this->queryGrammar = $this->getDefaultQueryGrammar();
    }
    // 根据默认实现设置请求后置处理器
    public function useDefaultPostProcessor()
    {
        $this->postProcessor = $this->getDefaultPostProcessor();
    }

```

文件 \Illuminate\Database\MySQLConnection.php

```

    // 获取默认的 SQL 语法实例
    protected function getDefaultQueryGrammar()
    {
        return $this->withTablePrefix(new QueryGrammar);
    }
    // 获取默认的结果处理实例
    protected function getDefaultPostProcessor()
    {
        return new MySqlProcessor;
    }

```

通过上面的源码可以看出，对应数据库连接实例的创建依然使用简单工厂模式实现，对默认支持的四个数据库分别提供了四个数据库连接类，用于封装底层不同的实现方法。这四个数据库连接类都继承于 \Illuminate\Database\Connection 类，该类为数据库连接提供统一的接口，对于特殊的部分需要分别设计，如数据库 SQL 语法实例和结果处理实例对于四种数据库是不同的，需要单独设计。

10.2.3 查询构造器的实现

查询构造器类（Illuminate\Database\Query\Builder 类）实例封装了数据库连接实例、请求语法实例和结果处理实例，这里类的实例提供了统一的接口方法供查询构造器实例使用。

下面是查询构造器实例化的部分代码：

文件 `\Illuminate\Database\Connection.php`

```
// 针对一个数据表创建一个查询构造器
public function table($table)
{
    $processor = $this->getPostProcessor();
    $query = new QueryBuilder($this, $this->getQueryGrammar(),
$processor);
    return $query->from($table);
}
```

文件 `\Illuminate\Database\Query\Builder.php`

```
// 创建查询构造器实例
public function __construct(ConnectionInterface $connection, Grammar
$grammar, Processor $processor)
{
    $this->grammar = $grammar;
    $this->processor = $processor;
    $this->connection = $connection;
}
// 设置查询构造器针对的数据表
public function from($table)
{
    $this->from = $table;
    return $this;
}
```

查询构造器实例是通过数据库连接实例的 `table()` 方法实现的，在其构造函数中可以看到对输入参数进行了类型约束，体现设计模式中的针对接口编程的原则。虽然这些接口的底层实现是不同的，但是接口功能是统一的，那么到达查询构造器这个层就可以对应用提供统一的数据库操作接口了。

10.2.4 查询构造器的使用

在查询构造器中已经实现了对底层数据库操作的封装，为上层应用提供了统一的数据库操作接口。重要的是，Laravel 框架通过查询构造器对 PDO 扩展进行了重新封装，使其提供的接口使用更加方便，主要表现在数据库操作不再需要专门设计 SQL 语句，而由底层的 SQL 语法规则实例来创建，对结果的处理也不用专门获取，而是由结果处理实例完成数据的处理后直接返回，因此，只需要调用查询构造器相应的方法就可以实现对数据库的增加、删除、修改和查询的操作。下面依然以 “`DB::table('users')->get();`” 为例，介绍通过查询构造器的 `get()` 函数完成数据库内容的查询功能，这里也分为两个阶段，即 SQL 语句准备阶段

和 SQL 语句执行阶段。

1. SQL 语句准备阶段

这个阶段主要将查询构造器对应的方法翻译成相应的 SQL 语句。这里将 SQL 语句划分为不同的片段，片段名称存储在 SQL 语法规则实例 (MySQLGrammar 类实例) 的 \$selectComponents 属性中，包括 'aggregate'、'columns'、'from'、'joins'、'wheres'、'groups'、'havings'、'orders'、'limit'、'offset' 和 'lock'，针对每个片段都有相应的函数进行处理。下面给出部分实现代码：

文件 \Illuminate\Database\Query\Builder.php

```
// 执行一个 select 请求命令，返回数据库查询结果处理后的最终结果
public function get($columns = ['*'])
{
    if (is_null($this->columns)) {
        $this->columns = $columns;
    }
    return $this->processor->processSelect($this, $this->runSelect());
}
// 通过连接执行 select 语句，获取数据库查询结果
protected function runSelect()
{
    return $this->connection->select($this->toSql(), $this->getBindings(),
!$this->useWritePdo);
}
// 获取 SQL 查询语句
public function toSql()
{
    return $this->grammar->compileSelect($this);
}
```

在查询构造器实例方法的调用过程中，会设置对应片段值，如代码“\$this->from = \$table;”和“\$this->columns = \$columns;”。针对本节中的实例，“\$table”值为“users”，“\$columns”值为“*”。SQL 语句准备阶段主要是通过 SQL 语法规则管理器解析 SQL 语句，这里通过 toSql() 函数实现，然后调用语法规则管理器的 compileSelect() 方法实现 SQL 语句的解析。

文件 \Illuminate\Database\Query\Grammars\Grammar.php

```
// 编译一个 select 请求为 SQL 语句，返回“select * from 'users'”
public function compileSelect(Builder $query)
{
    if (is_null($query->columns)) {
        $query->columns = ['*'];
    }
}
```



```

return trim($this->concatenate($this->compileComponents($query)));
}
// 翻译 select 语句中的必要组成部分
// 返回 array("columns"=>"select *", "from"=>"from 'users'")
protected function compileComponents(Builder $query)
{
    $sql = [];
    foreach ($this->selectComponents as $component) {
        if (!is_null($query->$component)) {
            $method = 'compile'.ucfirst($component);
            $sql[$component] = $this->$method($query, $query->$component);
        }
    }
    return $sql;
}
// 编译请求语句的“select *”部分，返回“select *”
protected function compileColumns(Builder $query, $columns)
{
    if (!is_null($query->aggregate)) {
        return;
    }
    $select = $query->distinct ? 'select distinct ' : 'select ';
    return $select.$this->columnize($columns);
}
// 编译请求语句的 from 部分，返回“from 'users'”
protected function compileFrom(Builder $query, $table)
{
    return 'from '.$this->wrapTable($table);
}
// 连接 SQL 语句数组中的片段并去除空格
protected function concatenate($segments)
{
    return implode(' ', array_filter($segments, function ($value) {
        return (string) $value !== '';
    }));
}

```

在对 SQL 语句解析过程中，首先需要根据查询构造器中设置的片段值构造对应的片段，如对于 select 查询语句会构造 select 片段，即通过 compileColumns() 函数实现“select \$columns”部分，而 \$columns 为默认值“*”，从而实现 select 查询语句中的“select *”部分，接着构造 from 片段，即通过 compileFrom() 函数实现查询语句中的“from \$table”部分，这里实现为“from users”，最后通过 concatenate() 函数将两个查询语句片段组装成一个查询语句返回，即生成 SQL 语句“select * from users”。

2. SQL 语句执行阶段

完成 SQL 语句的翻译后，接下来需要执行 SQL 语句，而执行 SQL 语句也是通过 PDO 类的 `prepare()` 方法和 `PDOStatement` 类的 `execute()` 方法，对处理结果的获取则通过 `PDOStatement` 类的 `fetchAll()` 方法，只是这部分内容被查询构造器的接口函数封装了，下面给出具体实现代码，了解一下查询构造器底层的实现细节。

文件 `\Illuminate\Database\Connection.php`

```
// 针对一个数据库执行一个 select 语句，其中 $query 就是上节中翻译的 SQL 语句
public function select($query, $bindings = [], $useReadPdo = true)
{
    return $this->run($query, $bindings, function ($me, $query, $bindings)
    use ($useReadPdo)
    {
        if ($me->pretending()) {
            return [];
        }

        $statement = $this->getPdoForSelect($useReadPdo)-
>prepare($query);
        $statement->execute($me->prepareBindings($bindings));
        return $statement->fetchAll($me->getFetchMode());
    });
}

// 执行一个 SQL 语句并记录执行的上下文
protected function run($query, $bindings, Closure $callback)
{
    $this->reconnectIfMissingConnection();
    $start = microtime(true);
    try {
        $result = $this->runQueryCallback($query, $bindings, $callback);
    }
    // 省略异常处理部分代码
    $time = $this->getElapsedTime($start);
    $this->logQuery($query, $bindings, $time);
    return $result;
}

// 执行 SQL 语句
protected function runQueryCallback($query, $bindings, Closure $callback)
{
    try {
        $result = $callback($this, $query, $bindings);
    }
    // 省略异常处理部分代码
    return $result;
}
```

```
}
// ...
}
```

SQL 语句的执行是通过 `select()` 函数实现的，在该函数中调用 `run()` 方法，而该方法的第三个参数是一个匿名函数，这个匿名函数是执行 SQL 语句获取数据库操作结果的核心部分。在这个匿名函数中，通过 `getPdoForSelect()` 函数获取前期数据库连接实例中封装的 PDO 实例，通过 PDO 实例的 `prepare()` 方法准备 SQL 语句并返回 `PDOStatement` 类实例，然后调用该实例的 `execute()` 方法执行 SQL 语句，最后通过 `fetchAll()` 函数获取数据库操作结果，返回后再通过结果处理器进行数据处理与封装并将最终结果返回。

10.2.5 查询构造器的数据库操作

通过上面四个小节以 “`DB::table('users')->get();`” 为例介绍了查询构造器实现的原理及底层实现方法，其实通过查询构造器可以实现对数据库的增加、删除、修改和查询等操作，虽然功能不同，但是原理是相同的，下面介绍部分操作的方法。

1. 增加信息

对数据库中的某个表增加数据主要有两个函数可以实现，分别是 `insert()` 和 `insertGetId()`，其中 `insert()` 函数接收数组数据，可以同时添加一条或多条数据，返回值为 `bool` 型，而 `insertGetId()` 函数也接收数组数据，但一次只能添加一条数据，返回值为添加数据行的自动递增 id 号。下面给出相关函数使用方法的实例。

向 “users” 数据表中添加一条数据：

```
DB::table('users')->insert([
    ['name' => 'shuoshuo', 'pass' => '222222']
]);
```

向 “users” 数据表中同时添加多条数据：

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

向 “users” 数据表中添加一条数据并获取自动递增的 id 号：

```
$id = DB::table('users')->insertGetId(['email' => 'shuoshuo', 'pass' =>
'212121'] );
```

2. 删除数据

数据删除可以通过 `delete()` 函数和 `truncate()` 函数实现，`delete()` 函数可以直接使用，也可以通过 `where()` 函数以链式方式添加删除条件，直接使用时是删除数据表中所有的数据，而添加条件后是删除所有符合条件的数据。`truncate()` 函数是清空数据表中的内容，也是删除数据表中的数据，结果和直接使用 `delete()` 函数相同，只是 `delete()` 函数是一条一条删除

数据的，而 `truncate()` 函数是释放数据表的空间，速度比 `delete()` 快得多，并且 `delete()` 函数的底层是使用 DML 语句来操作数据库，而 `truncate()` 函数的底层是使用 DDL 语句来操作数据库。下面给出两个函数操作数据库的具体实例。

删除 “users” 数据表中的数据：

```
DB::table('users')->where('id', '>', 10)->delete();
```

删除 “users” 数据表中的所有数据：

```
DB::table('users')->delete();
```

清空 “users” 数据表：

```
DB::table('users')->truncate();
```

3. 修改数据

数据的修改可以使用 `update()`、`increment()` 和 `decrement()` 等函数实现，其中 `update()` 函数也可以通过 `where()` 函数以链式形式添加修改条件，输入参数为修改后的数据，以数组的形式给出。`increment()` 和 `decrement()` 函数是对一个字段的值进行增加或减少，如果只设置字段，则自增或自减 1，如果设置字段和值，则增加和减少相应的数值。下面给出相应函数操作数据库的具体实例。

更新 “users” 数据表中的一条数据：

```
DB::table('users')->where('id', 2)->update(['pass' => "111111"]);
```

自增或自减 “users” 数据表中的一个字段的值，分别增加或减少 1 或 10：

```
DB::table('users')->increment('score');
```

```
DB::table('users')->increment('score', 10);
```

```
DB::table('users')->decrement('score');
```

```
DB::table('users')->decrement('score', 10);
```

4. 查询数据

数据库的查询主要通过 `get()` 函数实现，数据库的操作最为复杂的就是查询，为获取准确的数据，需要设计合适的查询条件，而 Laravel 框架的查询构造器为查询条件的设置设计了很多接口，可以通过链式方式添加这些条件，如 `where()`、`whereBetween()` 和 `whereIn()` 等函数，如果对 SQL 语句有所了解，根据这些函数名就可以了解添加条件的基本内容。下面给出相应函数操作数据库的具体实例。

获取 “users” 数据表中所有的数据：

```
$users = DB::table('users')->get();
```

获取满足 `where` 条件的数据：

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

获取满足 where 或 orWhere 条件的数据:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

获取在 whereBetween 区间中的数据:

```
$users = DB::table('users')->whereBetween('votes', [1, 100])->get();
```

获取在 whereIn 值中的数据:

```
$users = DB::table('users') ->whereIn('id', [1, 2, 3])->get();
$users = DB::table('users') ->whereNotIn('id', [1, 2, 3])->get();
```

10.3 Eloquent ORM

在 Laravel 框架中, 提供了另一个数据库扩展, 即 Eloquent ORM, 该扩展实现了 Active Record 模式与数据库进行交互, 使得操作数据库变得极为简单。这里先从概念上理解 Eloquent ORM 是什么。ORM 英文是 Object Relational Mapping (对象关系映射), 这是一种面向对象编程中用于解决不同系统间数据转换的方案, 相当于创建了一个“虚拟对象数据库”, 通俗的理解就是将数据库中复杂的结构封装成更加容易使用的接口提供给用户。在 Laravel 中, Eloquent ORM 就是实现这样的功能。而 Active Record 模式也遵循标准的 ORM 模型, 即数据库中每个表对应一个类, 而类的实例对应于数据表中的一行记录, 数据表中的字段映射到实例对象的属性。根据上述准则设计模型类, 从而完成对数据表的增加、删除、修改和查询等操作。

10.3.1 Eloquent ORM 的底层实现

在 Laravel 框架中, Eloquent ORM 扩展的底层依然使用的是查询构造器实现的, 这里通过简单对比两者的区别来理解 Active Record 模式的思想。在查询构造器中访问数据表时需要通过 table() 函数给定表名, 而在 Eloquent ORM 中每个类就对应一个数据表; 查询构造器查询结果返回的是数据数组, 而在 Eloquent ORM 中返回的则是类的实例对象, 每个实例对象对应一行数据; 在 Eloquent ORM 中类直接封装了对该数据表的操作, 使用时更加方便。下面依然通过一个简单的实例来介绍 Eloquent ORM 的底层实现, 然后通过一个实例来介绍模型类中封装的方法。

1. 模型类的创建

下面介绍模型类的创建和操作。一般模型类存放在 laravel/app 目录下, 当然也可以根据

实际需求放置在其他位置，只要符合 composer 自动载入的规范就可以。这里在 laravel/app 目录下建立一个 User 类并继承自 Model 类，这样一个 Eloquent ORM 模型就创建了，通常也称它为模型类，\$timestamp 属性用于表示 Laravel 框架自定义时间字段无效，因为在设计表的时候没有用这个字段。就这样简单，一个模型类就创建完成了，虽然你觉得自己什么都没做，但是这个类已经与一个数据表生成了对象关系映射，通过操作这个模型类，就可以操作对应的数据表了。下面先通过这个模型类获取表中所有的数据，来看看它到底是怎么工作的，只需要通过 “User::all();” 语句就可以实现上述功能。具体实例代码如下：

文件 laravel/app/User.php

```
<?php namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
    public $timestamps = false;
}
```

文件 laravel/app/Http/Controllers/UserController.php

```
<?php namespace App\Http\Controllers;
use App\User;
use App\Http\Controllers\Controller;
class UserController extends Controller
{
    public function index()
    {
        $data = User::all();
        var_dump($data);
    }
}
```

2. 模型类的实现原理

在新创建的模型类中其实什么都没有做，就可以实现对数据库操作的相关功能，而这些功能都是在继承类 Model 中完成的。但是，没有无条件的便利，因为底层是事先实现好的，在使用时就需要遵守它的规则，那么需要对底层的实现原理有一个清晰的认识才能更好地去应用。下面就通过上一节中介绍的实例，逐步了解模型类实现的流程，这里也将它分为两个阶段：第一阶段是 Eloquent ORM 查询构造器的生成，第二阶段是操作命令的执行。本实例是查询数据表中的所有数据。首先，介绍第一阶段，这里给出部分代码细节：

文件 \Illuminate\Database\Eloquent\Model.php

```
// 从数据表中获取所有模型
public static function all($columns = ['*'])
```



```

{
    $instance = new static;
    return $instance->newQuery()->get($columns);
}
// 创建一个新的 Eloquent 模型类实例
public function __construct(array $attributes = [])
{
    $this->bootIfNotBooted();
    $this->syncOriginal();
    $this->fill($attributes);
}

```

这里首先通过“new static;”语句生成一个模型类实例，用到了后期静态绑定，生成 App\User 类的实例，在本实例中，因为没有传递任何参数，所以构造函数并没有执行有意义的工作，这部分将在后期封装数据时再来介绍。对于后期静态绑定可以在 PHP 的重要性质中了解相关内容。接下来将通过“\$instance->newQuery()”语句生成 Eloquent ORM 查询构造器。

文件 \Illuminate\Database\Eloquent\Model.php

```

// 针对模型类对应的数据表生成一个新的查询构造器
public function newQuery()
{
    $builder = $this->newQueryWithoutScopes();
    return $this->applyGlobalScopes($builder);
}
// 获取查询构造器
public function newQueryWithoutScopes()
{
    $builder = $this->newEloquentBuilder($this->newBaseQueryBuilder());
    return $builder->setModel($this)->with($this->with);
}
// 获取针对一个连接的查询构造器
protected function newBaseQueryBuilder()
{
    $conn = $this->getConnection();
    $grammar = $conn->getQueryGrammar();
    return new QueryBuilder($conn, $grammar, $conn->getPostProcessor());
}
// 通过模型类获取数据库连接
public function getConnection()
{
    return static::resolveConnection($this->connection);
}

```

```
// 获取一个连接实例
public static function resolveConnection($connection = null)
{
    return static::$resolver->connection($connection);
}
```

文件 \Illuminate\Database\DatabaseManager.php

```
// 获取一个数据库连接的实例
public function connection($name = null)
{
    list($name, $type) = $this->parseConnectionName($name);
    if (!isset($this->connections[$name])) {
        $connection = $this->makeConnection($name);
        $this->setPdoForType($connection, $type);
        $this->connections[$name] = $this->prepare($connection);
    }

    return $this->connections[$name];
}

// 为模型类创建一个新的 Eloquent 查询构造器
public function newEloquentBuilder($query)
{
    return new Builder($query);
}
```

文件 \Illuminate\Database\Eloquent\Builder.php

```
public function __construct(QueryBuilder $query)
{
    $this->query = $query;
}
```

前文中介绍到，Eloquent ORM 的底层使用了查询构造器来实现，这里通过 `newQueryBuilder()` 函数创建一个基础查询构造器，而这个基础查询构造器中的数据库连接最终也是通过数据库控制器 (Illuminate\Database\DatabaseManager 类实例) 的 `connection()` 函数实现的。同时，SQL 语法规则实例和结果处理实例与上一小节中介绍的也是相同的，所以对于这个基础查询构造器无论是对应的类还是内部封装的相关属性都与上一小节介绍的查询构造器相同。在完成基础查询构造器的实例化后，通过 `newEloquentBuilder()` 函数对该查询构造器进行封装，实现 Eloquent 查询构造器的创建。

文件 \Illuminate\Database\Eloquent\Builder.php

```
// 为模型查询构造器设置一个模型实例
public function setModel(Model $model)
{
}
```

```
$this->model = $model;
$this->query->from($model->getTable());
return $this;
}
```

文件 \Illuminate\Database\Query\Builder.php

// 设置所要查询的数据表

```
public function from($table)
{
    $this->from = $table;
    return $this;
}
```

文件 \Illuminate\Database\Eloquent\Model.php

// 获取模型关联的数据表

```
public function getTable()
{
    if (isset($this->table)) {
        return $this->table;
    }

    return str_replace('\\', '', Str::snake(Str::plural(class_
basename($this))));
}
```

在完成 Eloquent 查询构造器的实例化后，也就获取了与数据库的连接、SQL 语法规则和结果处理的功能，Eloquent ORM 与上一节查询构造器操作数据库不同的是，这里还需要封装其他部分，包括模型类实例和数据表名称等，主要是需要封装模型类实例，因为对数据库的操作不再以数组的形式交互，而是通过这个模型类实例来与数据库进行交互。数据表名称是通过 Model 类对象的 getTable() 函数获取的，如果对象中的 \$table 属性存在值，则使用该数据表名，如果不存在，则以模型类名的复数作为数据表名，即通过代码 “str_replace('\\', '', Str::snake(Str::plural(class_basename(\$this))))” 实现，这就是为什么创建的模型类没有指定数据表名，依然可以操作数据库中的数据表的原因，在本实例中，默认的数据表名为 “users”。

3. 对数据库的操作

通过上一小节知道 Eloquent 查询构造器为 \Illuminate\Database\Eloquent\Builder 类的实例，而该实例又封装了查询构造器 \Illuminate\Database\Query\Builder 类的实例。在完成 Eloquent 查询构造器的实例化后，将实现对数据库的操作，即 “\$instance->newQuery()->get(\$columns);” 中的 get() 方法部分。接下来将介绍 Eloquent 查询构造器中数据库操作方法实现的原理。

文件 \Illuminate\Database\Eloquent\Builder.php


```
// 执行一个 select 查询语句
public function get($columns = ['*'])
{
    $models = $this->getModels($columns);
    if (count($models) > 0) {
        $models = $this->eagerLoadRelations($models);
    }
    return $this->model->newCollection($models);
}
```

在调用 “User::all()” 查询语句时首先是调用 Model 类的 all() 方法，然后构造 Eloquent 查询构造器 (Eloquent\Builder 类实例，注意与 Query\Builder 类区别)，从而调用该实例的一个 select 查询语句的 get() 方法来实现。在 SQL 语言中使用 select 查询语句来完成所有的数据库查询任务，这里用 get() 方法只实现了查询中常用的一部分功能，即 “select \$columns from \$table”，而 \$columns 和 \$table 表示要查询的数据字段名和数据表名。这个过程分为两个步骤：一是完成数据的查询，二是实现数据的封装。下面将介绍数据是如何被查询的及数据最终被封装成什么形式。

文件 \Illuminate\Database\Eloquent\Builder.php

```
// 获取模型类实例的集合
public function getModels($columns = ['*'])
{
    $results = $this->query->get($columns);
    $connection = $this->model->getConnectionName();
    return $this->model->hydrate($results, $connection)->all();
}
```

在 getModels() 函数中实际上完成了数据的查询，即通过 “\$this->query->get(\$columns);” 获取查询数据，这里的 \$this->query 实际上封装的是查询构造器类实例，通过查询构造器的 get() 方法获取查询数据，数据的形式是数组。也就是说，对于数据库的增加、删除、修改和查询等功能的底层实现是依赖查询构造器来实现的，而查询构造器对数据库增加、删除、修改和查询等功能的底层实现是依赖 PDO 扩展来实现的，就这样通过一层层封装不断为用户提供更加优秀的数据库操作接口。接下来介绍 Eloquent ORM 是如何封装结果数据的，这部分是通过 hydrate() 实现的。

文件 \Illuminate\Database\Eloquent\Model.php

```
// 为一个数组创建一个模型类实例的集合
public static function hydrate(array $items, $connection = null)
{
    $instance = (new static)->setConnection($connection);
    $items = array_map(function ($item) use ($instance) {
```

```

        return $instance->newFromBuilder($item);
    }, $items);
    return $instance->newCollection($items);
}
// 创建一个新的模型类实例
public function newFromBuilder($attributes = [], $connection = null)
{
    $model = $this->newInstance([], true);
    $model->setRawAttributes((array) $attributes, true);
    $model->setConnection($connection ?: $this->connection);
    return $model;
}
// 通过数组设置模型类实例的 attributes 属性
public function setRawAttributes(array $attributes, $sync = false)
{
    $this->attributes = $attributes;
    if ($sync) {
        $this->syncOriginal();
    }
}
// 创建一个新的 Eloquent 集合实例
public function newCollection(array $models = [])
{
    return new Collection($models);
}

```

文件 Illuminate\Support\Collection.php

```

// 创建一个 \Illuminate\Support\Collection 类实例
public function __construct($items = [])
{
    $this->items = is_array($items) ? $items : $this-
>getArrayableItems($items);
}

```

获取查询数据后将对数据进行封装，通过 `array_map()` 函数对数组中的每项进行处理，处理函数为一个匿名函数，该匿名函数中调用 `newFromBuilder()` 函数进行数据处理，该函数通过 `newInstance()` 函数实例化一个 `Model` 类，并将数据赋值给该实例的 `$attributes` 属性。接着实例化一个 `Illuminate\Database\Eloquent\Collection` 类实例，该类继承自集合类 `Illuminate\Support\Collection`，并将 `array_map()` 函数生成的模型类实例数组传递给集合类构造函数，最终将查询获取的数据以 `Eloquent` 集合的形式进行封装，而 `Eloquent` 集合中存储的是一个模型类实例数组，每一个模型类实例对应查询的一条结果。

10.3.2 Eloquent ORM 的使用

前面已经介绍了 Eloquent ORM 的实现原理，下面介绍它的使用。由于数据库操作的内容非常多，因此无法做到面面俱到，这里通过一个实例来介绍一些常用方法的使用，如果在应用中需要了解更多可以查看官方手册。

1. 数据表的创建与数据填充

为了介绍 Eloquent ORM 的使用，首先需要构建数据表并添加相应的数据。要完成数据表的创建和数据填充一般需要三个步骤：一是在数据库中创建对应的数据表，二是对应每个数据表创建模型类文件，三是完成数据库的填充。在数据表的创建和数据填充过程中，可以使用 artisan 命令来创建相应的数据迁移文件。数据表创建时可以通过“php artisan make:migration create_blogs_table --create=blogs”命令创建“blogs”数据表，同理创建其他的数据表迁移文件，然后通过“php artisan migrate”命令完成数据库中数据表的创建。在创建数据表模型类时可以使用“php artisan make:model Blog”命令来创建对应的模型类。在实现数据填充时可以通过“php artisan db:seed”命令执行数据填充。这部分内容可以参考本章第一节的内容，下面分别介绍三个步骤的实现细节和实现代码。

(1) 建立几个数据表。

```
Schema::create('blogs', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('title');
});
Schema::create('author', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
    $table->integer('blog_id');
});
Schema::create('comments', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('content');
    $table->integer('words');
    $table->integer('blog_id');
});
Schema::create('subjects', function(Blueprint $table)
{
    $table->increments('id');
    $table->string('name');
```



```
Schema::create('blogs_subjects', function(Blueprint $table)
{
    $table->increments('id');
    $table->integer('blog_id');
    $table->integer('subject_id');
});
```

这里使用数据库迁移创建了 5 个表, 第一个是文章表 (blogs 表), 第二个是作者表 (authors 表), 第 3 个是评论表 (comments 表), 第 4 个是专题表 (subjects 表), 第五个是文章与专题关系表 (blogs subjects 表)。每个表除了记录相应内容外, 还记录了表之间的关系, 这里的关系假设如下: 一篇文章只能有一个作者, 一个作者只能发表一篇文章, 即一对一关系; 一篇文章有多条评论, 即一对多关系; 一篇文章可以在多个专题中, 一个专题可以记录多篇文章, 即多对多关系。这里涉及到的对应关系分别是一对一关系、一对多关系和多对多关系, 其中多对多关系在 blogs_subjects 表中进行记录。

(2) 根据数据表建立 Eloquent 模型类。

```
class Blog extends Model
{
    // 以下字段可以被批量赋值
    protected $fillable = array('title');
    // 不使用 Laravel 提供的默认时间
    public $timestamps = false;
    // 定义文章模型关系, 一篇文章只能有一个作者
    public function author()
    {
        return $this->hasOne('App\Author');
    }
    // 一篇文章有多条评论
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
    // 一篇文章可以在多个专题中, 一个专题可以包含多篇文章
    public function subjects()
    {
        return $this->belongsToMany('App\Subject', 'blogs_
subjects', 'blog_id', 'subject_id');
    }
}

class Author extends Model
{
    protected $fillable = array('name', 'blog_id');
    // 新的表名, 不使用默认 authors 表名
```

```

protected $table = 'author';
public $timestamps = false;
// 定义关联关系
public function blog()
{
    return $this->belongsTo('App\Blog');
}

class Comment extends Model
{
    protected $fillable = array('comment', 'words', 'blog_id');
    public $timestamps = false;
    public function blog()
    {
        return $this->belongsTo('App\Blog');
    }
}

class Subject extends Model
{
    protected $fillable = array('name');
    public $timestamps = false;
    public function blogs()
    {
        return $this->belongsToMany('App\Blog', 'blogs_subjects',
'subject_id', 'blog_id');
    }
}

```

上文提到，可以使用“php artisan make:model Blog”命令来创建对应的模型类，这里定义了前四个表的模型类，而关系数据表在模型类中用于构建关系，通过 \$fillable 定义可以批量赋值内容，批量赋值就是通过模型类的构造函数将数据表中的一行数据一次性赋值，实际上是通过 Model 类的 fill() 函数来完成的，当然也可以通过对模型类的属性依次直接赋值。通过 \$table 定义数据表的名称，如果没有定义则会使用默认数据表名。通过 hasOne()、hasMany()、belongsTo() 和 belongsToMany() 函数可以定义表间的关系。

(3) 为数据表填充数据。

创建数据填充类 BlogSeeder 并继承自 Seeder 类，改写 run() 方法。需要注意的是，在执行 artisan 命令时，实际上执行的是 Laravel 默认数据填充文件 DatabaseSeeder.php 中的 run() 方法，需要在该方法的 call() 方法中添加创建的填充文件，即添加代码“\$this->call(BlogSeeder::class);”。

文件 laravel/database/seeds/BlogSeeder.php

```
<?php
```

```

use App\Blog;
use App\Author;
use App\Comment;
use App\Subject;
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class BlogSeeder extends Seeder {
    public function run() {
        // 清空所有表数据
        DB::table('blogs')->delete();
        DB::table('author')->delete();
        DB::table('comments')->delete();
        DB::table('subjects')->delete();
        DB::table('blogs_subjects')->delete();
        // 填充 blogs 表, 创建三篇文章
        $php = Blog::create(array(
            'title' => 'PHP 的未来 '    ));
        $java = Blog::create(array(
            'title' => 'Java 的未来 '    ));
        $html5 = Blog::create(array(
            'title' => 'HTML5 的未来 '    ));
        // 命令行文字提示
        $this->command->info('The blogs are seeded!');
        // 填充 author 表
        Author::create(array(
            'name' => 'PHP 的作者 ',
            'blog_id' => $php->id
        ));
        Author::create(array(
            'name' => 'Java 的作者 ',
            'blog_id' => $java->id
        ));
        Author::create(array(
            'name' => 'HTML5 的作者 ',
            'blog_id' => $html5->id
        ));
        // 填充 comments 表
        Comment::create(array(
            'content' => 'PHP 多用于服务器程序编写 ',
            'words' => 13,
            'blog_id' => $php->id
        ));
        Comment::create(array(

```



```

        'content' => 'PHP 是无类型编程语言 ',
        'words'    => 11,
        'blog_id' => $php->id
    ));
    // 填充 subjects 表
    $language = Subject::create(array(
        'name'=> ' 计算机语言 ',
    ));
    $program = Subject::create(array(
        'name'=> ' 编程语言 ',
    ));
    // 关联 blogs 与 subjects
    $php->subjects()->attach($language->id);
    $php->subjects()->attach($program->id);
    $java->subjects()->attach($language->id);
    $java->subjects()->attach($program->id);
    $html5->subjects()->attach($language->id);
}
}

```

这里在文件数据表（blogs 表）中添加三篇文章，题目（title 字段）分别为“PHP 的未来”、“Java 的未来”和“HTML5 的未来”，为了简化文章没有添加内容。在作者表（author 表）中添加了三个作者，分别是“PHP 的作者”、“Java 的作者”和“HTML5 的作者”，记录在 name 字段中，同时在 blog_id 字段记录发表的文章 id 号。在评论表（comments 表）中添加两条评论，都是对“PHP 的未来”文章的评论，即在 blog_id 字段中记录了该文章在数据表中的 id 号。在专题表（subjects 表）中添加两个专题，分别是“计算机语言”和“编程语言”，其中 PHP 和 Java 既属于计算机语言也属于编程语言，而 HTML5 只属于计算机语言。对于上面的实例可能有些与实际不符，这里只是假设这样。

2. CRUD 操作的实现

前面已经完成了数据表的建立、模型类的创建和数据库中数据的填充，接下来将实现数据库的简单操作。这里对数据库的操作不仅包括数据的增加、删除、修改和查询，还包括对数据表间关系的查询。下面给出具体实例代码。

（1）在 blogs 数据表中增加新数据。

1) 通过 create() 函数以批量赋值的方式实现数据添加。

```
Blog::create(array('title'=> 'Python 的未来 ' ));
```

2) 通过属性赋值和 save() 方法实现数据添加。

```

$blog= new Blog;
$blog->title= 'Ruby 的未来 ';
$blog->save(); // 增加数据

```

3) 通过 `firstOrCreate()` 或 `firstOrCreateNew()` 方法增加数据, 这两个函数首先查询相关记录, 如果不存在才会去创建。

```
// 查询或创建记录
Blog::firstOrCreate(array('title' => 'JavaScript 的未来'));
// 查找并生成实例
$blog = Blog::firstOrCreate(array('title' => 'CSS3 的未来'));
$blog->save();
```

需要注意的是, 虽然 `firstOrCreate()` 和 `firstOrCreateNew()` 两个方法都是查询数据, 如果不存在就创建新的记录, 但在创建新记录时还是存在差别的, `firstOrCreate()` 方法是创建数据模型实例并存入数据库, 而 `firstOrCreateNew()` 方法只创建了数据模型实例, 还没有添加进数据库, 需要再执行一次 `save()` 函数, 才能实现数据的增加。

(2) 查询相应数据表中的数据。

1) 获得所有记录。

```
$blogs = Blog::all();
```

2) 通过 id 查询数据。

```
$blog = Blog::find(1);
```

3) 通过指定字段名查询数据。

```
$blog = Blog::where('title', '=', 'PHP 的未来')->first();
```

4) 查询字数大于 10 个的评论数据。

```
$commentss = Comment::where('words', '>', 10)->get();
```

5) 查询字数大于 10 个小于 15 个的评论数据。

```
$commentss = Comment::whereRaw("words>10 or words<15")->get();
```

(3) 更新数据表中的记录。

1) 通过 Eloquent ORM 更新一条记录首先要查询到它, 然后更改数据模型实例的属性, 最后调用 `save()` 方法完成数据更新。

```
// 将内容为“PHP 是无类型编程语言”的评论修改为“PHP 变量名以 $ 开头”, 并将评论字数由 11 变为 10
// 先查询到它
$comment = Comment::where('content', '=', 'PHP 是无类型编程语言')->first();
// 修改属性
$comment->content = "PHP 变量名以 $ 开头";
$comment->words = 10;
// 保存
$comment->save();
```

2) 通过 `update()` 函数实现模型更新。

```
$affectedRows = Comment::where('content', '=', 'PHP 是无类型编程语言')->update(array('content' => 'PHP 变量名以 $ 开头'));
```

(4) 删除数据表中的记录。

删除记录可以通过 `delete()` 方法或 `destroy()` 方法实现, 其中 `delete()` 方法为实例方法, 需要查询到相应的数据并通过模型实例调用, 而 `destroy()` 方法可以直接调用, 通过索引删除记录。

1) 通过 `delete()` 方法删除记录。

```
// 查找并删除记录
$blog = Blog::find(6);
$blog->delete();
// 删除字数多余 10 个的评论
Comment::where('words', '>', 10)->delete();
```

2) 通过 `destroy()` 方法删除记录。

```
// 删除 id 为 7 的单条记录
Blog::destroy(7);
// 删除 id 为 1、2、3 的三条记录
Blog::destroy(1, 2, 3);
```

3. 数据表间关系的查询

在实际项目开发中, 很多数据表都不是孤立存在的, 而是与其他数据表存在一定的关系, 如一对一关系、一对多关系和多对多关系, 而 Eloquent 可以根据定义直接进行关系查询, 获取与本数据表相关联的数据。

(1) 一对一关系查询。

在上述数据表中, 文章表中的数据与作者的关系为一对一关系, 下面通过文章字段查找作者的相关数据信息。实例代码如下:

```
// 查找标题为“PHP 的未来”的文章
$php= Blog::where('title', '=', 'PHP 的未来')->first();
// “PHP 的未来”的作者
$auth = $php->author;
// 作者的名字
$auth->name;
// 也可以使用更快捷的方法来获得作者的名字
$auname = $php->author->name;
echo " 文章“PHP 的未来”的作者 :". $auname. "</br>";
```

输出结果:

文章“PHP 的未来”的作者 :PHP 的作者

对于数据表的关系查询, 首先要在模型类中定义相应的关系, 如在 Blog 模型类中通过 `author()`、`comments()` 和 `subjects()` 函数分别定义一对一、一对多和多对多关系, 当查询到数

据表中的一条数据时，该数据被封装在一个模型类实例中，于是可以通过属性的形式来查询对应关系的数据，即通过代码“\$php->author”查询作者的信息。

(2) 一对多关系。

在上述例子中，文章和评论是一对多的关系，如标题为“PHP 的未来”的文章有两条评论，内容分别是“PHP 多用于服务器程序编写”和“PHP 是无类型编程语言”。下面给出一对多关系查询的实例代码：

```
// 找到标题为“PHP 的未来”的文章
$php = Blog::where('title', '=', 'PHP 的未来')->first();
// 获取该文章的评论并输出
foreach ($php->comments as $comment)
echo $comment->content . ' ' . $comment->words.'

```

(3) 多对多关系。

在上述实例中，文章和专题是多对多关系，一篇文章可以属于多个专题，一个专题可以拥有多篇文章。这里简单地假设两个专题，分别是计算机语言和编程语言，其中标题为“PHP 的未来”、“Java 的未来”和“HTML5 的未来”都属于计算机语言专题，而只有“PHP 的未来”和“Java 的未来”属于编程语言专题。下面给出多对多关系查询的实例代码：

```
// 查找标题为“PHP 的未来”的文章
$blog = Blog::where('title', '=', 'PHP 的未来')->first();
// 查看该文章属于的专题
echo " 文章“PHP 的未来”属于专题: </br>";
foreach ($blog->subjects as $subject){
    echo $subject->name.'

```

输出结果：

文章“PHP 的未来”属于专题：

计算机语言

编程语言

专题“编程语言”包含文章：

PHP 的未来

Java 的未来

Laravel 框架中关于 Eloquent ORM 的内容很多, 因为篇幅原因无法全部详细介绍, 这里只介绍了部分使用方法及实现的原理, 如果需要了解更多可以查看源码和官方文档。

本章从 PHP 对数据库扩展开始, 重点介绍了 Laravel 框架中查询构造器和 EloquentORM 的底层实现。数据对于一个应用就像心脏对于人一样能够为整个应用带来活力, 其性能的好坏直接决定了应用程序的性能, 而很多情况下数据库的性能是受不当的增加、删除、修改和查询影响的, 所以在使用这些模块带来便利的同时, 也需要深入地了解底层的实现, 否则可能会设计出功能齐全、性能低下的应用。就拿 ORM 来说, ORM 映射最大的好处是将数据表的结构映射成一个类对象, 那么就可以将数据以对象的形式进行封装使用, 程序的编写将变得高效而且结构清晰, 但是 ORM 映射的使用有时也存在困难, 对于单个数据表来说问题不大, 但是对于多个表而且表间存在不同的关系时, 采用 ORM 映射查询就需要注意了, 如果使用不好会严重影响程序的性能, 这就需要开发人员将底层的实现机制了解清楚, 在进行数据库操作的时候哪些操作会严重影响性能要有认识, 这样才能搭建出一个高性能的应用。

11.1.2 redis 数据库结构

11.1 redis 数据库简介

Redis 数据库是一种开源的内存数据库, 它支持多种数据类型, 如字符串、列表、集合、有序集合、哈希等。Redis 数据库的特点包括: 高性能、高可用、持久化、支持分布式部署等。Redis 数据库广泛应用于缓存、消息队列、分布式锁等场景。Redis 数据库的架构如下:

Redis 数据库的架构分为三层: 客户端、服务器和持久化层。客户端通过 TCP 连接与服务器通信, 服务器负责处理客户端的请求, 并将数据存储在内存中。持久化层负责将内存中的数据持久化到磁盘上, 以防止数据丢失。

Redis 数据库的持久化方式有两种: RDB 和 AOF。RDB 是 Redis 数据库的默认持久化方式, 它通过定期将内存中的数据快照保存到磁盘上。AOF 是 Redis 数据库的另一种持久化方式, 它通过记录客户端的每一个写操作来持久化数据。

Redis 数据库的持久化层位于服务器的本地磁盘上, 它通过 Redis 进程的配置文件进行配置。Redis 数据库的持久化层还支持分布式部署, 可以通过 Redis 集群的方式进行部署。

第 11 章

redis 数据库

Redis 数据库是一个开源的数据库，既提供了操作数据的内存存储，又提供了数据的磁盘存储。redis 数据库一般会将存入的数据先存在内存中，当存储的内容达到一定数量或者经过一定时间后，才将内容存储到磁盘上，所以 redis 数据库对数据的存储和操作非常快，因为大部分数据操作是在内存中完成的。同时，redis 数据库要比 MemCache 功能更强，因为它可以实现数据的持久化存储，即存储在磁盘上，而且 redis 数据库提供了 strings、hashes、lists、sets、sorted sets、bitmaps 和 hyperloglogs 七种数据结构，其中 hyperloglogs 数据结构是在 redis 数据库的 2.8.9 版本后支持的，而 bitmaps 数据结构是在 3.0 版本后提供的，这两种数据结构都是为大数据处理而设计的，如果要使用这两种类型的数据结构，需要安装相应版本的 redis 数据库，以提供相应的程序接口。Laravel 框架是通过添加“predis/predis”资源包来实现与 redis 数据库的交互，该资源包目前只支持前六种数据结构。下面对 redis 数据库进行简单介绍，然后介绍在 Laravel 框架中通过 redis 数据库可以实现的不同功能。

11.1 redis 数据库简介

11.1.1 安装

Redis 数据库的官网是 <http://redis.io/>，在 redis 官网中的软件下载部分提到 redis 只支持 Linux 操作系统。对于 Windows 系统，redis 官网声明不提供官方 Windows 版本的 redis，不过微软开源科技组 (Microsoft Open Tech group) 开发并维护了 Win64 版本的 redis 数据库，可以到网址 <https://github.com/MicrosoftOpenTech/redis> 查看相关内容。Linux 操作系统下安装 redis 比较简单，只需要通过命令“`apt-get install redis-server`”就可以完成。在完成 redis 安装后，可以通过命令“`nohup redis-server &`”实现 redis 数据库服务端的启动，通过“&”使其在后台启动，并在当前目录下的“nohup”文件中记录 redis 服务端的输出内容。通过命令“`cat nohup`”可以查看到 redis 服务端启动输出的信息，如图 11.1 所示，表示目前安装的 redis 版本为 2.8.4，PID 号为 1316。接着可以通过命令“`redis-cli`”启动 redis 数据库的客户端，启

动后就可以通过命令行操作 redis 数据库了，如图 11.2 所示。

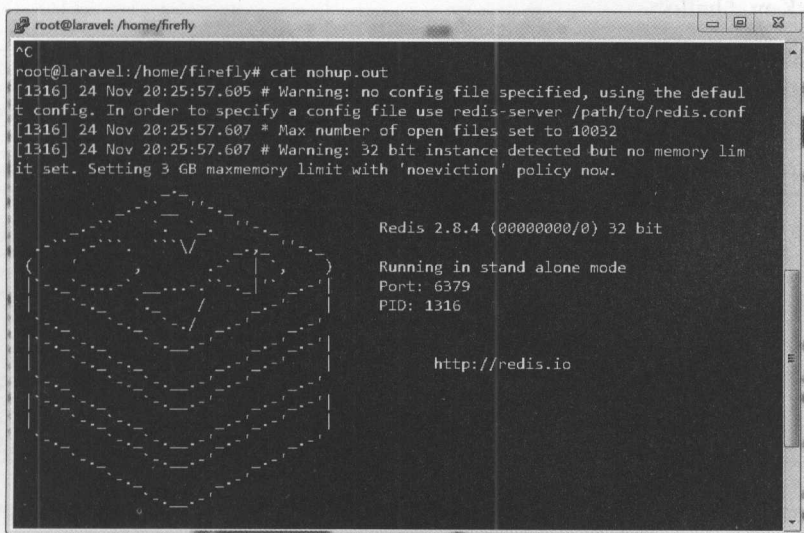


图 11.1 redis 服务端启动输出内容

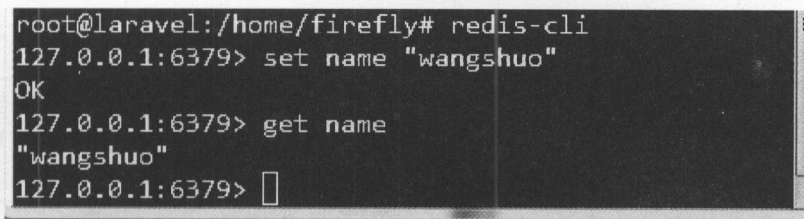


图 11.2 redis 客户端命令行操作

11.1.2 redis 数据结构

Redis 数据库是一款依赖 BSD 开源协议的高性能 Key-Value 数据存储系统，通常又被称为数据结构服务器，因为它原生地支持了几种常用的数据结构，给程序设计带来相当大的便利和性能提升。目前 redis 最新版本为 3.0.5 版本，支持七种类型的数据结构，其中 bitmaps 和 hyperloglogs 是在后期版本中加入的，是为大数据处理而提供的。下面对 redis 数据库的操作做简单的介绍，使读者有一个初步的认识，这样读者就可以学习 Laravel 中关于 redis 的内容了。

1. key 的操作

redis 数据库都是以“键值对”的形式存储数据的，数据值可以有不同的数据类型和数据结构，对于键来说都是字符串，可以通过操作相应的键来操作对应的数据值。同时，redis 数据库也提供了丰富的键操作接口命令，这里首先在 redis 数据库中添加以下数据，接着使用相应的数据库操作命令得到对应结果，具体如表 11.1 所示。

设置的键：

SET newKey "hello"

SET timeKey "time"

SET testKey 10

表 11.1 键值操作功能表

操 作	功 能	实 现	结 果
KEYS KEYS pattern	查找符合 pattern 模式的 key	KEYS t*	返回 “testKey” 和 “timeKey”
EXISTS EXISTS key	查询一个 key 是否存在	EXISTS testKey	返回 1，表示存在
TYPE TYPE key	获取 key 存储元素的类型	TYPE testKey	返回 string，表示字符串类型
EXPIRE EXPIRE key seconds	设置 key 的生存时间，单位为秒	EXPIRE timeKey 10	返回 1，表示设置成功； 返回 0，表示设置失败
TTL TTL key	获取 key 的剩余时间，单位为秒	TTL timeKey	返回 8，即有效时间为 8 秒
RENAME RENAME key newkey	将一个 key 重命名为 newkey	RENAME testKey otherKey	返回状态 ok，可以通过 GET 获取 otherKey 的值
DEL DEL key [key...]	移除一个或多个指定的 key	DEL otherKey	返回 1，表示删除成功

2. 字符串 (String) 数据结构

对于字符串数据结构来说，每个键对应一个字符串的值，如果是数值可以进行普通的加减，也可以进行相应的位操作。下面介绍几个常用的字符串操作命令，如表 11.2 所示。

表 11.2 字符串 (String) 数据结构操作功能表

操 作	功 能	实 现	结 果
SET SET key value	设置一个 key 的 value 值	SET testKey “hello”	通过 GET testKey 获取 “hello” 值
GET GET key	获取 key 的值	GET testKey	返回 “hello”
INCR INCR key	将 key 键对应存储的数字值加 1	SET numKey “10” INCR numKey	返回 11
DECR DECR key	将 key 键对应存储的数字值减 1	DECR numKey	返回 10

续表

操 作	功 能	实 现	结 果
MSET MSET key value [key value...]	同时设置一个或多个 键值对	MSET tKey1 “hello” tKey2 “world”	返回 ok 状态，表示设 置成功
SETBIT SETBIT key index value	设置字符串一个 bit 位的值	SETBIT bitKey 5 1	返回 0，即之前第 5 位 的值为 0，现在已经设置 为 1
GETBIT GETBIT key index	返回相应 bit 位上的 值	GETBIT bitKey 5	返回 1
BITCOUNT BITCOUNT key	统计字符串中设置为 1 的 bit 位的总数	BITCOUNT bitKey	返回 1，表示只有一个 bit 位被设置为 1

3. 哈希 (Hash) 数据结构

对于哈希数据结构来说，相应键上对应的值为一个哈希结构，相当于散列表，只是该散列表中对应的值只能是字符串或数值类型，不能是对象。下面介绍几个常用的哈希操作命令，如表 11.3 所示。

表 11.3 哈希 (Hash) 数据结构操作功能表

操 作	功 能	实 现	结 果
HSET HSET key field value	设置哈希 key 里面的一 个字段 field 的值为 value	HSET userHash name “lily”	返回 1，表示设置了 userHash 新的字段
HGET HGET key field	获取哈希 key 字段 field 的值	HGET userHash name	返回 “lily”
HMSET H M S E T key field value [field value...]	同时设置哈希 key 的一 个或多个字段值	HMSET userHash2 name “lily” age 18	返回 “ok” 状态码， 表示设置成功
HMGET HMGET key field [field..]	获取哈希 key 的一个或 多个字段值	HMGET userHash2 name age	返回 “lily” 和 “18”
HGETALL HGETALL key	获取哈希 key 的所有字 段和值	HGETALL userHash2	返回 “name”、“lily”、 “age” 和 “18”
HVALS HVALS key	获取哈希 key 的所有值	HVALS userHash2	返回 “lily” 和 “18”
HDEL HDEL key field [field...]	删除一个或多个哈希 key 的字段 field	HDEL userHash2 name age	返回 2，表示成功删 除两个字段

4. 列表 (List) 数据结构

利用列表数据结构，可以实现诸如队列、栈等一些高级的特性，如消息队列就可以使用列表来实现。列表分为列表头和列表尾，列表的左端或前端称为列表头，列表的右端或后端称为列表尾。下面介绍几种常用的列表操作命令，如表 11.4 所示。

表 11.4 列表 (List) 数据结构操作功能表

操 作	功 能	实 现	结 果
LPUSH LPUSH key value [value ...]	将一个或多个值插入到列表头	LPUSH testList "hello"	返回 1，表示列表长度目前为 1
RPUSH RPUSH key value [value ...]	将一个或多个值插入到列表尾	RPUSH testList "world"	返回 2，表示列表长度目前为 2
LLEN LLEN key	返回列表的长度	LLEN testList	返回 2，表示列表 testList 长度为 2
LRANGE LRANGE key start stop	获取指定列表的指定区间的值	LRANGE testList 0 -1	返回 "hello" 和 "world"
LINSERT LINSERT key BEFORE AFTER pivot value	将值插入到列表中并位于值 pivot 之前或之后	LINSERT testList BEFORE "world" "the"	返回 3，将在 "world" 值之前插入 "the"
LINDEX LINDEX key index	获取列表下标为 index 的值	LINDEX testList 0	返回 "hello"，返回列表的第一个值
LSET LSET key index value	将列表下标为 index 的值设置为 value	LSET testList 1 "all"	将列表中第二个元素值 "the" 修改为 "all"
LPOP LPOP key	弹出列表左边 / 头元素	LPOP testList	返回 "hello"，移除列表中的第一个元素 "hello"
RPOP RPOP key	弹出列表右边 / 尾元素	RPOP testList	返回 "world"，移除列表中最后一个元素 "world"

5. 集合 (Set) 数据结构

集合数据结构要求集合内的值不能重复，同时集合内的元素没有顺序之分，集合之间可以实现交、差、并等关系运算。下面介绍几种常用的集合操作命令，如表 11.5 所示。

表 11.5 集合 (Set) 数据结构操作功能表

操 作	功 能	实 现	结 果
SADD SADD key member	将元素加入到集合中	SADD testSet "a" SADD testSet "b" SADD testSet "c"	返回 1，表示插入成功。这里向集合中成功插入三个元素，即 "a"、"b"、"c"

续表

操 作	功 能	实 现	结 果
SMEMBERS SMEMBERS key	返回集合中的所有元素	SMEMBERS testSet	返回 testSet 集合的全部元素，即“a”、“b”、“c”
SISMEMBER SISMEMBER key mem	判断元素 mem 是否在集合内	SISMEMBER testSet “a”	返回 1，表示元素“a”在集合 testSet 中
SCARD SCARD key	返回集合的基数	SCARD testSet	返回 3，表示集合 testSet 中有 3 个元素
SMOVE SMOVE src des mem	将元素 mem 从 src 集合中移动到 des 集合中	SMOVE testSet otherSet “b”	返回 1，表示“b”元素成功从 testSet 集合中移除，添加进 otherSet 集合中
SRANDMEMBER SRANDMEMBER key	返回集合中一个随机元素	SRANDMEMBER testSet	返回“c”，返回一个随机元素，集合不变
SUNION SUNION key [key…]	返回一个或多个集合的并集	SUNION testSet otherSet	返回“a”、“c”、“b”，即集合 testSet(“a”、“c”)和 otherSet(“b”)的并集
SUNIONSTORE SUNIONSTORE des key [key…]	将返回的集合并集保存到 des 集合中	SUNIONSTORE threeSet testSet otherSet	返回 3，将 testSet 和 otherSet 集合并集保存到 threeSet 中，threeSet 中的元素为“a”、“c”、“b”
SDIFF SDIFF key [key…]	返回所有集合差集的所有元素	SDIFF threeSet testSet	返回“b”，即两个集合的差集
SINTER SINTER key [key…]	返回所有集合的交集	SINTER testSet threeSet	返回“a”、“c”，即两个集合的交集

前面主要讨论了 redis 数据库的键的操作，并对四种数据结构（字符串、哈希、列表和集合）的模型和操作进行了简单的介绍。下面对其他三种数据结构模型进行简单介绍。有序集合 (Sorted Set) 实际上是在集合的基础上为每个元素添加的权重，使得集合元素具有了顺序，权重小的在前 / 左，权重大的在后 / 右。位图 (bitmap) 可以看做是大型的字符串，可变字符串由于在长度增加时需要重新开辟一块空间，所以将原来的字符串复制到当前位置时要增加一定的空间，但大的字符串的复制很占内存，如果复制频率过高则严重影响性能，特别是大数据处理这种情况，而使用 bitmap 可以一次分配很大的连续空间，最大为 512MB，这样可以存储很大的数据而不存在大数据的复制问题。hyperloglogs 是专门为大数据存储设计的一种数据结构，相当于对数据进行压缩存储，普通的数据通过 hyperloglogs 存储后占用的空间相当于原来占用空间的百分之一左右，对于大数据的存储和处理有很重要的意义。这就是 redis 数据库关于数据结构的基本内容，发布、订阅等功能在 Laravel 框架的 redis 应用中再进行介绍。

11.2 redis 数据库的应用

在 Laravel 框架中使用 redis 数据库需要满足两点要求，一是安装 redis 数据库，redis 数据库的安装和使用在上一节中已经进行了介绍；二是需要通过 composer 安装 “predis/predis” 资源包，版本为 “~1.0”。这里可以在 Laravel 框架根目录下执行 composer require “predis/predis”:“~1.0” 命令实现资源包的安装，其中 “~1.0” 是赋值运算版本号，匹配的版本为 ($\geq 1.0, < 2.0$)。通过 composer 的 require 命令将需求包的信息添加到根目录下的 composer.json 中，同时下载资源包并更新 autoload 文件。也可以修改根目录下的 composer.json 文件，添加 “predis/predis” 资源包后，通过 composer update 命令下载并更新 autoload 文件。需要注意的一点是，使用 “predis/predis” 资源包不需要安装 PHP5 的 redis 扩展，因为该资源包直接实现了 redis 数据库的客户端功能而没有使用 PHP5 的 redis 扩展。如果在系统中安装了 PHP5 的 redis 扩展，那么在使用 redis 数据库时不能直接通过 “use redis;” 方式使用 redis 数据库的外观 (Facades, 有的书翻译成门面)，而是需要使用完整的命名空间名称，即通过 “use Illuminate\Support\Facades\redis” 方式使用 redis 数据库的外观，否则将会与 redis 扩展命名空间冲突，导致外观方式使用失败。下面进一步介绍 Laravel 框架中 redis 数据库的使用。

11.2.1 数据存取

根据前面的介绍，redis 数据库是在内存中操作数据，所以对数据的读取操作效率更高，同时由于非关系型数据库可以根据键值直接索引数据，因此数据的查找更加快捷，其编程思想也与关系型数据库有很大的差别。下面介绍 Laravel 框架对 redis 数据库中数据的存取操作。首先给出 Laravel 框架操作 redis 数据库程序的路由和控制器代码，具体如下：

路由文件 laravel \app\Http\routes.php

```
<?php
Route::get('gl', 'WebController@index');
```

控制器文件 laravel \app\Http\Controllers\WebController.php

```
<?php
namespace App\Http\Controllers;
// 如果安装了 PHP5 的 redis 扩展，需要使用下面的命名空间
//use Illuminate\Support\Facades\Redis;
use Redis;
use App\Http\Controllers\Controller;
class WebController extends Controller
{
    public function index()
    {
        Redis::set('string:user:name', 'wshuo');
```



```
echo Redis::get('string:user:name');
```

这里通过代码“Redis::set('string:user:name','wshuo');”向 redis 数据库中的“string:user:name”键中保存字符串“wshuo”，然后通过代码“Redis::get('string:user:name');”获取 redis 数据库中“string:user:name”键的字符串。通过 Laravel 的外观，可以直接以“Redis::”方式访问 redis 数据库，并通过“predis/predis”资源包实现对数据库中数据的存储和查询，其中方法名和 redis 数据库命令行使用的命令相同，最终使得对 redis 数据库的操作变得非常简单而直观。redis 数据库操作的命令有很多，Laravel 框架在整合了 predis 资源包后将这些操作的过程划分为三个阶段：第一阶段是以外观方式通过服务容器获取 redis 数据库客户端服务；第二阶段是 redis 数据库客户端实例化过程；第三阶段是操作指令的生成和发出。接下来进一步讲解这三个阶段的实现细节。

1. 通过外观获取服务

通过 Laravel 框架外观获取服务容器中的相应服务在前面的章节中已经介绍过了，这里只是简要地进行介绍。对于上述实例，通过外观获取服务的过程相当于代码的“Redis::”部分，Redis 类实际是 Illuminate\Support\Facades\Redis 类，只是以别名的方式使用，该类继承了 Façade 类，get() 和 set() 方法需要调用 Façade 类中的魔术方法 __callStatic() 来实现。具体实现代码细节如下：

文件 Illuminate\Support\Facades\Facade.php

```
// 处理一个对象的动态或静态方法
public static function __callStatic($method, $args)
{
    $instance = static::getFacadeRoot();
    switch (count($args)) {
        case 0:
            return $instance->$method();
        case 1:
            return $instance->$method($args[0]);
        // 省略部分 case 情况的代码
    }
}
```

文件 Illuminate\Support\Facades\Redis.php

```
// 获取注册组件的名称
protected static function getFacadeAccessor()
{
    return 'redis';
}
```

文件 Illuminate\Support\Facades\Facade.php

// 通过服务容器解决外观对应的实例

```
protected static function resolveFacadeInstance($name)
{
    if (is_object($name)) {
        return $name;
    }

    if (isset(static::$resolvedInstance[$name])) {
        return static::$resolvedInstance[$name];
    }

    return static::$resolvedInstance[$name] = static::$app[$name];
}
```

所谓的外观其实就是定义了一些外观类，如 Illuminate\Support\Facades\Redis 类，而该类只做一件事，就是提供服务容器中相应服务的名称，如 redis 服务，同时通过配置文件 config/app.php 将外观类定义了别名 “Redis” (“Redis’ => Illuminate\Support\Facades\Redis::class,”), 所以当通过 “Redis::set()” 这种外观形式调用一个静态方法时，实际上是获取服务容器中相应服务名称的实例，并调用该实例的方法。

2. redis 数据库客户端的实例化

通过前面的介绍，了解了使用 “Redis::” 方法实际上是获取服务容器中对应的服务，即 static::\$app['redis']，而该服务是由服务提供者注册的一个回调函数，该回调函数用于实现 redis 数据库客户端的实例化。下面介绍实例化过程的相关代码：

文件 Illuminate\Redis\RedisServiceProvider.php

// 服务提供者注册服务

```
public function register()
{
    $this->app->singleton('redis', function ($app) {
        return new Database($app['config']['database.redis']);
    });
}
```

文件 Illuminate\Redis\Database.php

// 创建一个 redis 连接实例

```
public function __construct(array $servers = [])
{
    $cluster = Arr::pull($servers, 'cluster');
    $options = (array) Arr::pull($servers, 'options');
    if ($cluster) {
        $this->clients = $this->createAggregateClient($servers, $options);
    } else {
```

```
$this->clients = $this->createSingleClients($servers, $options);
```

文件 Illuminate\Redis\Database.php

// 创建单连接客户端的一个连接数组

```
protected function createSingleClients(array $servers, array $options = [])
{
    $clients = [];
    foreach ($servers as $key => $server) {
        $clients[$key] = new Client($server, $options);
    }
    return $clients;
}
```

这里通过服务容器获取 redis 的服务是由服务提供者 Illuminate\Redis\RedisServiceProvider 类注册的，通过服务回调函数返回一个 \Illuminate\Redis\Database 类实例，这里将其称为 redis 数据库客户端管理器类。在构造函数中需要提供 redis 数据库配置信息 (\$app['config']['database.redis']), 包括主机地址、端口号和数据库编号等，默认情况下 redis 数据库不配置认证密码，如果需要可以手动配置，而且 redis 数据库名称是以数字编号区分的，如 0、1、2 等，相应的配置信息记录在 “config/database.php” 配置文件中。在 redis 数据库客户端管理器实例化过程中，如果配置参数 “\$cluster” 设置为 true，则生成的客户端实例以一个数组的形式保存在关联数组的 default 键中，默认情况下 “\$cluster” 设置为 false，即每个关联数组项只有一个客户端实例。Laravel 框架配置文件中定义了一个客户端实例配置项 (default 项)，对于大型项目可能需要多台 redis 数据库服务器，就会创建一个多客户端的关联数组，通过 “\$clients” 属性进行保存。下面介绍一个 redis 数据库客户端的实例化过程。

文件 laravel\vendor\predis\predis\src\Client.php

```
public function __construct($parameters = null, $options = null)
{
    $this->options = $this->createOptions($options ?: array());
    $this->connection = $this->createConnection($parameters ?: array());
    $this->profile = $this->options->profile;
}
```

文件 laravel\vendor\predis\predis\src\Client.php

// 根据不同的参数创建一个 predis 的配置选项实例

```
protected function createOptions($options)
{
    if (is_array($options)) {
        return new Options($options);
    }
}
```



```

    }
    // 省略其他方式生成配置选项实例代码部分
}

```

文件 laravel \vendor\predis\predis\src\Configuration\Options.php

```

public function __construct(array $options = array())
{
    $this->input = $options;
    $this->options = array();
    $this->handlers = $this->getHandlers();
}

```

文件 laravel \vendor\predis\predis\src\Configuration\Options.php

```

// 返回默认的选项实例的初始化配置
protected function getHandlers()
{
    return array(
        'cluster' => 'Predis\Configuration\ClusterOption',
        'connections' => 'Predis\Configuration\ConnectionFactoryOption',
        'exceptions' => 'Predis\Configuration\ExceptionsOption',
        'prefix' => 'Predis\Configuration\PrefixOption',
        'profile' => 'Predis\Configuration\ProfileOption',
        'replication' => 'Predis\Configuration\ReplicationOption',
    );
}

```

一个 redis 数据库客户端实例实际上就是一个 predis\Client 类的实例，该实例中主要记录了三个属性，分别为配置项、连接和概况 (\$options、\$connection 和 \$profile)，其中配置项为 predis\Configuration\Options 类实例，该类实例中主要记录一个处理数组，即 \$handlers 数组，该数组记录了不同模块的处理类，所以可以称其为模块处理类管理表，当需要实现某个功能时会调用相应模块的函数获取实例对象。上面的代码即为配置项类的实例化过程，下面介绍连接实例的创建过程。

文件 laravel \vendor\predis\predis\src\Client.php

```

// 根据不同的参数创建一个单独的连接
protected function createConnection($parameters)
{
    if ($parameters instanceof ConnectionInterface) {
        return $parameters;
    }

    if ($parameters instanceof ParametersInterface || is_string($parameters)) {
        return $this->options->connections->create($parameters);
    }
}

```

```

    }
    if (is_array($parameters)) {
        if (!isset($parameters[0])) {
            return $this->options->connections->create($parameters);
        }
    }
    // 省略一部分代码
}

```

文件 laravel \vendor\predis\predis\src\Configuration\Options.php

// 创建连接实例对象

```

public function __get($option)
{
    // 此时参数 $option="connections", 这里省略其他方式生成连接代码部分
    if (isset($this->handlers[$option])) {
        return $this->options[$option] = $this->getDefault($option);
    }
    return;
}

```

文件 laravel \vendor\predis\predis\src\Configuration\Options.php

// 创建连接实例对象

```

public function getDefault($option)
{
    if (isset($this->handlers[$option])) {
        $handler = $this->handlers[$option];
        $handler = new $handler();
        return $handler->getDefault($this);
    }
}

```

文件 laravel \vendor\predis\predis\src\Configuration\ConnectionFactoryOption.php

// 连接工厂实例创建

```

public function getDefault(OptionsInterface $options)
{
    return new Factory();
}

```

连接的实例化过程可以分为三个步骤：第一步是通过配置选项实例完成连接工厂实例的创建；第二步是创建连接实例；第三步是连接实例的配置。前两个步骤由代码“`$this->options->connections->create($parameters);`”实现，连接工厂实例的创建相当于“`$this->options->connections`”代码部分，其中“`$this->options`”为前期完成的配置选项实例，由于该实例中没有“connections”属性，所以会调用 `__get()` 魔术方法，该函数会在配置选项实例中的“\$handlers”数组属性（前文提到的模块处理类管理表）中查找，于是得到连接处

理类 `Predis\Configuration\ConnectionFactoryOption` 并进行实例化，然后通过连接处理类实例的 `getDefault()` 方法获取默认的连接创建工厂实例，即 `\predis\Connection\Factory` 类实例。接下来介绍客户端连接实例化过程。

文件 `laravel\vendor\predis\predis\src\Connection\Factory.php`

// 连接实例创建

```
public function create($parameters)
{
    if (!$parameters instanceof ParametersInterface) {
        $parameters = $this->createParameters($parameters);
    }
    $scheme = $parameters->scheme;
    // 省略异常处理代码部分
    $initializer = $this->schemes[$scheme];
    if (is_callable($initializer)) {
        $connection = call_user_func($initializer, $parameters, $this);
    } else {
        $connection = new $initializer($parameters);
        $this->prepareConnection($connection);
    }
    // 省略异常处理部分代码
    return $connection;
}
```

上述代码主要实现 redis 数据库客户端实例中最主要的一个属性的创建，即连接属性，是通过连接工厂的 `create()` 函数实现的。在创建连接实例前需要先获取参数实例，即 `\predis\Connection\Parameters` 类实例，该实例对 redis 配置文件中的参数进行了过滤并添加了新的参数，即 “`scheme=> 'tcp'`”，表示客户端实例是以 “tcp” 方式与服务端进行连接，在 `\predis\Connection\Factory` 类实例的 `$schemes` 属性中记录了不同连接方式的实现类，然后通过该参数获取 “tcp” 连接方式的实现类为 `Predis\Connection\Stream Connection` 类，最后通过代码 “`new $initializer($parameters)`” 生成该类的实例，即实现一个连接的实例化过程。

文件 `laravel\vendor\predis\predis\src\Connection\Factory.php`

// 在连接实例初始化后进行完善

```
protected function prepareConnection(NodeConnectionInterface $connection)
{
    $parameters = $connection->getParameters();
    if (isset($parameters->password)) {
        $connection->addConnectCommand(
            new RawCommand(array('AUTH', $parameters->password))
        );
    }
}
```



```

        if (isset($parameters->database)) {
            $connection->addConnectCommand(
                new RawCommand(array('SELECT', $parameters->database))
            );
        }
    }
}

```

文件 laravel\vendor\predis\predis\src\Command\RawCommand.php

// 原始命令实例创建

```

public function __construct(array $arguments)
{
    // 省略异常处理部分代码
    $this->commandID = strtoupper(array_shift($arguments));
    $this->arguments = $arguments;
}

```

文件 laravel\vendor\predis\predis\src\Connection\AbstractConnection.php

// 添加命令实例到初始化命令数组中

```

public function addConnectCommand(CommandInterface $command)
{
    $this->initCommands[] = $command;
}

```

接下来将完成连接实例创建的最后一个步骤，即完成连接实例的配置，主要是判断 redis 数据库是否需要密码认证及数据库的选择，其中密码是在“password”配置项中设置的，数据库是在“database”配置项中设置的，Laravel 框架默认 redis 数据库是不设置密码的，同时数据库选择“0”。这里需要创建数据库选择的命令，即生成一个 Predis\Command\RawCommand 类实例，成为原始命令实例，实例中包括命令 ID 号和命令参数，对于数据库选择命令实例来说，命令 ID 号为“SELECT”，命令参数为“0”。生成原始命令实例后会添加到命令数组中，等待客户端初始化完毕后进行发送。

在经过完善后，redis 数据库客户端实例中的连接属性已经构建完成，接下来完成概况的实例化，它记录了客户端所支持的操作，不同版本的 redis 数据库所支持的操作命令是不同的，所以要针对不同的版本设置相应的客户端，记录该版本支持的操作命令，当通过客户端发送指令时，首先需要通过概况来过滤该指令是否符合版本要求，符合的指令才会通过客户端实例中的连接发送出去。下面是 redis 数据库客户端概况实例的代码细节。

文件 laravel\vendor\predis\predis\src\Configuration\Options.php

```

// 对应的代码是“$this->profile = $this->options->profile”，参数 $option = profile
public function __get($option)
{

```

```
// 省略部分代码，用于实现生成的实例不再重新生成
    if (isset($this->handlers[$option])) {
        return $this->options[$option] = $this->getDefault($option);
    }
    return;
}
```

文件 `laravel \vendor \predis \predis \src \Configuration \Options.php`

// 获取默认 profile 的实例，即 `$handler = "Predis \Configuration \ProfileOption"` 类

```
public function getDefault($option)
{
    if (isset($this->handlers[$option])) {
        $handler = $this->handlers[$option];
        $handler = new $handler();
        return $handler->getDefault($this);
    }
}
```

文件 `laravel \vendor \predis \predis \src \Configuration \ProfileOption.php`

// 创建概况实例对象

```
public function getDefault(OptionsInterface $options)
{
    $profile = Factory::getDefault();
    $this->setProcessors($options, $profile);
    return $profile;
}
```

文件 `laravel \vendor \predis \predis \src \Profile \Factory.php`

// 返回默认的客户端概况实例

```
public static function getDefault()
{
    return self::get('default');
}
```

文件 `laravel \vendor \predis \predis \src \Profile \Factory.php`

// 返回指定的客户端概况实例

```
public static function get($version)
{
    if (!isset(self::$profiles[$version])) {
        throw new ClientException("Unknown server profile: '$version'.");
    }
    $profile = self::$profiles[$version];
    return new $profile();
}
```

文件 `laravel\vendor\predis\predis\src\Profile\RedisProfile.php`

// 创建概况实例

```
public function __construct()
{
    $this->commands = $this->getSupportedCommands();
}
```

文件 `laravel\vendor\predis\predis\src\Profile\RedisVersion300.php`

// 获取支持的操作命令

```
public function getSupportedCommands()
{
    return array(
        'EXISTS' => 'Predis\Command\KeyExists',
        'DEL' => 'Predis\Command\KeyDelete',
        'TYPE' => 'Predis\Command\KeyType',
        'KEYS' => 'Predis\Command\KeyKeys',
    );
}
```

// 省略部分操作指令代码

客户端概况的实例化过程与连接的实例化过程相似，也是通过代码“`$this->options->profile`”获取模块处理类管理表中关于概况的处理类，即 `Predis\Configuration\ProfileOption` 类，通过该类实例的 `getDefault()` 函数获取概况工厂类，在概况工厂类的“`$profiles`”属性中记录了不同版本 redis 数据库的概况类，默认情况先选择版本“`default`”，即概况类为“`Predis\Profile\RedisVersion300`”，最后通过概况工厂的 `get()` 函数实现概况的实例化。通过上述源码的实现过程，可以知道客户端的概况属性用于记录与版本支持相关的信息，主要是 redis 数据库的命令。在默认情况下，会生成 redis 数据库版本为 3.0 的概况实例，也就是支持 redis 3.0 数据库的所有操作命令。

至此，redis 数据库客户端的实例化过程就结束了，通过三个关键属性（配置选项、连接和概况）来完成客户端的功能。概况用来过滤生成命令，连接用来发送传输命令，配置选项用来记录相应模块的管理类，进行客户端实例化的整个过程控制。完成了客户端的实例化后，就可以准备发送命令操作数据库了。

3. 操作指令的生成与发送

在完成了客户端实例化后，可以简单地看做完成了代码“`Redis::set('string:user:name','wshuo');`”中的“`Redis::`”部分，实际上是实现了 `Illuminate\Support\Facades\Facade` 类中的 `__callStatic()` 函数的“`$instance = static::getFacadeRoot();`”部分，接下来将实现 `set()` 方法部分，实际上是实现 `__callStatic()` 函数中的“`$instance->$method($args[0], $args[1]);`”部分，其中 `$method="set"`，`$args[0]="string:user:name"`，`$args[1]="wshuo"`。操作指令的生成和发送过程分为两个阶段，即指令生成阶段和指令发送阶段。下面介绍实例生成阶段的实现过程。

文件 Illuminate\Redis\Database.php

```
// 动态生成一个 redis 指令
// 参数 $method='set', $parameters=array('string:user:name', ' wshuo')
public function __call($method, $parameters)
{
    return $this->command($method, $parameters);
}
```

文件 Illuminate\Redis\Database.php

```
// 针对 redis 数据库执行一条指令
public function command($method, array $parameters = [])
{
    return call_user_func_array([$this->clients['default'], $method],
    $parameters);
}
```

文件 laravel \vendor\predis\predis\src\Client.php

```
// 这里 $commandID="set", $arguments=array("string:user:name", "wshuo")
public function __call($commandID, $arguments)
{
    return $this->executeCommand(
        $this->createCommand($commandID, $arguments)
    );
}
// 创建一条命令
public function createCommand($commandID, $arguments = array())
{
    return $this->profile->createCommand($commandID, $arguments);
}
```

文件 laravel \vendor\predis\predis\src\Profile\RedisProfile.php

```
// 命令实例生成
// 此时参数 $ commandID = 'set', $ arguments =array('string:user:name', ' wshuo')
public function createCommand($commandID, array $arguments = array())
{
    $commandID = strtoupper($commandID);
    // 省略异常处理部分代码
    $commandClass = $this->commands[$commandID];
    $command = new $commandClass();
    $command->setArguments($arguments);
    if (isset($this->processor)) {
        $this->processor->process($command);
    }
    return $command;
}
```

操作指令首先需要完成指令实例的生成,在前面客户端实例生成过程中已经提到,指令的生成是由客户端实例中的概况属性负责实现的,根据操作的指令类型进行过滤,获取指令类名称,即由代码“`$commandClass = $this->commands[$commandID];`”实现,然后实例化该指令类,添加相应的指令参数,最终完成指令的实例化过程。接下来将指令通过客户端发送到服务端,在 `redis` 资源包中,指令的发送是通过 `redis` 通信协议完成的,而不是借助 `PHP` 中的 `redis` 扩展实现的。下面是代码实现的具体过程。

文件 `laravel\vendor\predis\predis\src\Client.php`

```
// 执行 redis 数据库命令
public function executeCommand(CommandInterface $command)
{
    $response = $this->connection->executeCommand($command);
    // 省略错误处理部分代码
    return $command->parseResponse($response);
}
```

文件 `laravel\vendor\predis\predis\src\Connection\AbstractConnection.php`

```
// 执行命令
public function executeCommand(CommandInterface $command)
{
    $this->writeRequest($command);
    return $this->readResponse($command);
}
```

文件 `laravel\vendor\predis\predis\src\Connection\StreamConnection.php`

```
public function writeRequest(CommandInterface $command)
{
    $commandID = $command->getId();
    $arguments = $command->getArguments();
    $cmdlen = strlen($commandID);
    $reqlen = count($arguments) + 1;
    $buffer = "{$reqlen}\r\n${$cmdlen}\r\n{$commandID}\r\n";
    for ($i = 0, $reqlen--; $i < $reqlen; ++$i) {
        $argument = $arguments[$i];
        $arglen = strlen($argument);
        $buffer .= "{$arglen}\r\n{$argument}\r\n";
    }
    $this->write($buffer);
}
```

在完成指令实例化后,将会实现指令的发送。指令的发送也可以分为两个步骤,一是

生成符合 redis 数据库通信协议的指令序列，二是获取 redis 数据库的连接资源（为了简单可以将其看做是一个 TCP 连接）并将指令序列写入连接中。指令序列先标识指令序列参数的个数，即由“*3”来标识，然后是相应的参数，参数需要先指定参数字符个数，通过 \$3、\$16 和 \$8 来标识，最后是相应的指令，其中各标识符间通过换行 (r\n) 隔开。本实例中生成的指令序列为“*3\r\n\$3\r\nSET\r\n\$16\r\nstring:user:name\r\n\$8\r\nnwshuo\r\n”。

文件 laravel\vendor\redis\redis\src\Connection\StreamConnection.php

// 符合 redis 通信协议的指令序列写入连接中

protected function write(\$buffer)

```
{
    $socket = $this->getResource();
    while (($length = strlen($buffer)) > 0) {
        $written = @fwrite($socket, $buffer);
        if ($length === $written) {
            return;
        }
        // 省略错误处理部分代码
        $buffer = substr($buffer, $written);
    }
}
```

文件 laravel\vendor\redis\redis\src\Connection\AbstractConnection.php

// 获取客户端和服务端间的连接资源

public function getResource()

```
{
    if (isset($this->resource)) {
        return $this->resource;
    }
    $this->connect();
    return $this->resource;
}
```

// 创建客户端和服务端间的连接

public function connect()

```
{
    if (!$this->isConnected()) {
        $this->resource = $this->createResource();
        return true;
    }
    return false;
}
```

文件 laravel\vendor\redis\redis\src\Connection\StreamConnection.php

// 创建客户端与服务端的连接资源

```
protected function createResource()
```

```
{
    switch ($this->parameters->scheme) {
        case 'tcp':
        case 'redis':
            return $this->tcpStreamInitializer($this->parameters);
        case 'unix':
            return $this->unixStreamInitializer($this->parameters);
        // 省略异常处理部分代码
    }
}
```

```
}
```

文件 `laravel\vendor\predis\predis\src\Connection\StreamConnection.php`

// 初始化一个 TCP 连接资源

```
protected function tcpStreamInitializer(ParametersInterface $parameters)
```

```
{
    if (!filter_var($parameters->host, FILTER_VALIDATE_IP, FILTER_FLAG_
IPV6)) {
        $uri = "tcp://$parameters->host:$parameters->port";
    } else {
        $uri = "tcp://[$parameters->host]:$parameters->port";
    }
    $flags = STREAM_CLIENT_CONNECT;
    if (isset($parameters->async_connect) && (bool) $parameters->async_
connect) {
        $flags |= STREAM_CLIENT_ASYNC_CONNECT;
    }
    if (isset($parameters->persistent) && (bool) $parameters->persistent)
    {
        $flags |= STREAM_CLIENT_PERSISTENT;
        $uri .= strpos($path = $parameters->path, '/') === 0 ? $path :
"/$path";
    }
    $resource = @stream_socket_client($uri, $errno, $errstr, (float)
$parameters->timeout, $flags);
    // 省略错误处理代码和一些连接参数配置的代码部分
    return $resource;
}
```

这部分代码实现了连接资源的构建及指令序列的发送。socket 连接资源的构建是通过代码 “`$socket = $this->getResource();`” 实现的，实际上最终是通过 PHP 公共函数 `stream_socket_client()` 来实现的，该函数是初始化一个连接远程目标的数据报文连接。socket 地址的定义是通过标准 URL 格式来实现的，对于标准的 `AF_INET` 类型 socket（如

TCP 或 UDP)，需要指定目标 IP 地址和端口号，本实例中给出的 socket 地址为“\$uri=tcp://127.0.0.1:6379”，通过该地址可以连接本地 redis 数据库的服务端，然后通过 fwrite() 公共函数将指令序列写入 socket 资源中，完成指令的发送。

至此，完成了一个 redis 数据库操作的整个流程，虽然 redis 数据库的指令很多，但其执行流程基本是不变的。Laravel 框架操作 redis 数据库是通过 predis 资源包来实现的，其中 predis 资源包实现对单个 redis 服务器客户端实例化及指令发送的全部过程，而 Laravel 框架对其进行了封装，通过封装使得对 redis 服务器的操作能力获得提升，主要表现在两个方面：一是通过 Illuminate\Redis\Database 类实例实现对 redis 客户端实例的管理，实现了 redis 数据库连接池的功能，只需要在第一次操作某个 redis 数据库时实例化该客户端，之后的操作就可以直接通过该实例获取 redis 客户端并进行指令发送；二是将 Illuminate\Redis\Database 类实例注册到服务容器中，可以通过外观的方式实现 redis 客户端实例的获取和操作，使得对 redis 数据库的操作更加简单和直观。

11.2.2 redis 数据库编程思想

Redis 数据库不仅是一种非关系型数据库，而且提供了数据结构的服务，使得对它的应用与通常的关系型数据库存在一定差别，对于程序的开发，只有符合它的编程方式才能最大地发挥它的效率。因为 redis 数据库是以“键值对”的形式存储数据的，所以数据的查找就不需要像关系型数据库那样进行值的查询，而是通过“键”以索引的形式来查找数据，这种数据的查找方式相对于关系型数据库的值查询要快很多。下面通过一个简单的用户登录实例来介绍，具体代码如下：

文件 laravel/app\Http\Controllers\RedisAuthController.php

```
<?php
namespace App\Http\Controllers;
use Redis;
use App\Http\Controllers\Controller;
class RedisAuthController extends Controller
{
    public function register()
    {
        if(isset($_POST["username"])){
            $username = $_POST["username"];
            $password = $_POST["password"];
            Redis::set("string:users:".$username, $password);
        }
    }
    public function login()
    {
```

```

$username = $_POST["username"];
if($_POST["password"]==Redis::get("string:users:".$username))
{
    echo "login success";
}
else{
    echo "failed";
}
}
}

```

本实例中的用户名和密码是以“键值对”的形式存储的，即键为“string:users:".\$username”，而值存储的为用户的登录密码“\$password”。在用户登录认证中，直接通过键“string:users:".\$username”来获取用户的密码并进行比对，如果相同则登录成功。如果用关系型数据库，则需要在用户表中搜索所有的用户名并进行比对，找到对应的用户名后再获取用户密码，这种查找方式相对于 redis 数据库直接通过键获取值的方式效率要低很多。

在实际应用中，因为 redis 在内存中操作数据，只有超过一定时间或存储的数据量大于一定程度时才会存储到磁盘中，所以数据并不是绝对安全的，停电或重启等情况会导致数据丢失，而对于用户注册登录这个模块是需要数据绝对安全的，所以在使用时需要将 redis 数据库和关系型数据库（如 MySQL 数据库）一起使用。下面以 MySQL 数据库为例，当用户注册时，首先在 MySQL 数据库中存储用户数据，如果存储成功，则在 redis 数据库中存储用户数据。当用户登录时，先通过 redis 数据库查找用户信息，如果能够查找到，则用该用户的信息进行认证；如果查找不到，则到 MySQL 数据库中查找用户信息；如果在 MySQL 数据库中查找到用户信息，则用该信息进行认证，同时向 redis 数据库中再存储一份该用户信息，下次该用户再次登录时就会在 redis 数据库中查找到该用户的信息。由于绝大多数的用户信息都可以在 redis 数据库中查找到，所以使用 redis 数据库的效率要比使用 MySQL 数据库效率高，同时 MySQL 数据库中有用户信息的记录，也不会存在数据不安全的情况。

11.2.3 发布、订阅消息

Redis 数据库除了用于快速操作数据外，还有一个重要的功能，即可以发布、订阅消息，通过对消息的发布和订阅可以实现消息队列的功能。redis 是通过 publish 和 subscribe 指令提供消息的发布和订阅的，通过订阅一个“队列”用于监听消息，当有消息通过 publish 指令发布到“队列”中时，订阅程序就会监听到消息并进行响应。为了实现上述功能，可以使用 artisan 命令“php artisan make:console SubscribeMsg --command=Sub:Msg”来创建一个订阅消息类，并修改该类的 handle() 函数，通过 Redis::subscribe() 方式订阅一个“队列”，这里定义的队列名称为“redis-msg”。但是，只创建一个控制台指令类还是不行的，需要将该类在控制台核心类 (App\Console\Kernel) 的“\$commands”属性中进行注册才能生效。具

体实现代码如下：

文件 `laravel\app\Console\Commands\SubscribeMsg.php`

```
<?php    namespace App\Console\Commands;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis as Redis;
class SubscribeMsg extends Command
{
    // 控制台指令的名称
    protected $signature = 'Sub:Msg';
    // 控制台指令的描述
    protected $description = 'Redis Subscribe Command';
    public function __construct()
    {
        parent::__construct();
    }
    // 执行控制台命令
    public function handle()
    {
        Redis::subscribe(['redis-msg'], function($message) {
            echo $message;
        });
    }
}
```

文件 `laravel\app\Console\Kernel.php`

```
// 应用中提供的 artisan 命令
protected $commands = [
    \App\Console\Commands\Inspire::class,
    \App\Console\Commands\SubscribeMsg::class,
];
```

通过上述两个步骤，就实现了 `artisan` 命令的创建，其中命令名称为“`Sub:Msg`”，是通过 `SubscribeMsg` 类的“`$signature`”属性定义的，这个名称也可以随意修改，在“`$description`”属性中定义了该命令的注释。在创建和注册完命令后就可以通过 `artisan` 命令“`php artisan list`”来查看命令列表中的该命令，得到的部分命令列表如图 11.3 所示。

通过上述方式定义了一个 `artisan` 命令，并在命令中启用了 `redis` 订阅功能，它将持续监听“`redis-msg`”队列是否有消息，如果有消息则被发送到该队列中，通过回调函数将消息内容打印出来。在 `Laravel` 框架中，`redis` 消息的发送实现起来非常简单，通过 `Redis::publish()` 方式在程序运行的任何位置都可以实现消息的发送。下面设计发送一个网站首页被访问的消息。具体实现代码如下：

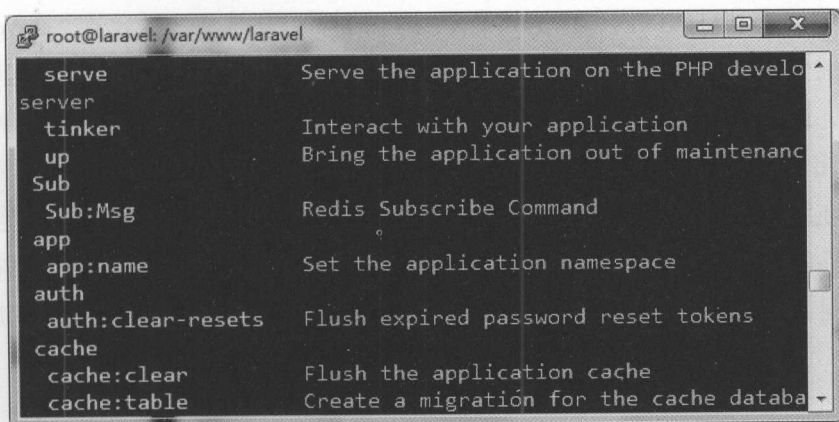


图 11.3 artisan 命令列表中自定义命令的显示

文件 laravel/app/Http/routes.php

```

<?php
Route::get('/', function () {
    Redis::publish('redis-msg', 'visit welcom time='.time()."");
    return view('welcome');
});

```

通过上述代码可以看到，只需要在路由文件的访问网站首页的路由函数中加入一条 redis 消息发送代码就可以实现该功能，其中发送的队列名称为“redis-msg”，需要发送的队列名称和订阅接收的队列名称相同，发送的消息为一行字符串，在实际中需要传输数据时一般会将其序列化为字符串或以 json、XML 等格式进行发送。为了验证实现效果，需要启用订阅功能，可以通过 artisan 命令“php artisan Sub:Msg”实现，然后通过浏览器访问本网站的首页触发消息发送功能，控制台即可显示监听到的消息结果，如图 11.4 所示。

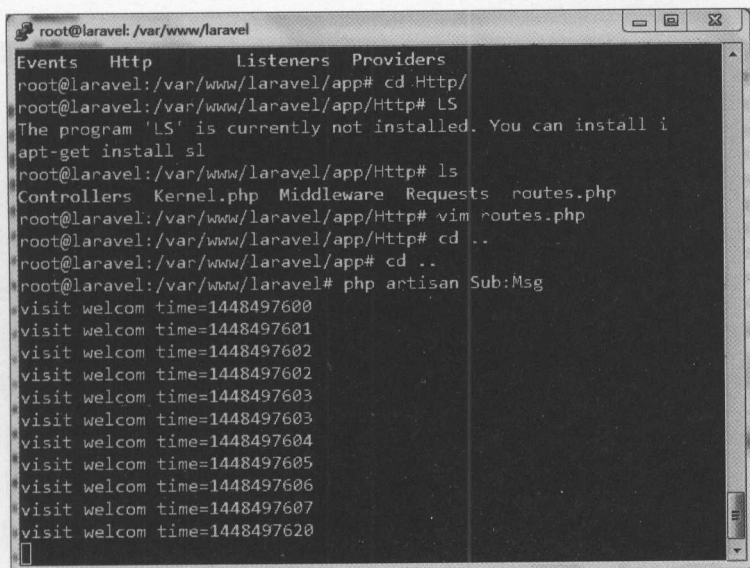


图 11.4 控制台显示监听到的消息结果

redis 消息发布和订阅的内容这里只是介绍了一种简单的实现方式，在 Laravel 官方文档上也有相应的介绍，底层的实现细节其实和前文中介绍的数据存储部分相似，这里就不进行展开说明了，读者可以按照前文数据存储的实现流程阅读 Laravel 源码进一步了解。

本章介绍了在 Laravel 框架中操作 redis 数据库的内容。redis 数据库作为非关系型数据库的一种，在结构性数据的存储、查询等方面非常高效。互联网项目大多以数据为驱动，构建的数据库性能直接决定了项目的性能，不同的非关系型数据库都在某些方面弥补了关系型数据库的不足。所以，要构建一个高效的项目，需要尽量考虑非关系型数据库的应用。

HTTP 协议是建立在 TCP/IP 协议上的一种无状态的请求 / 响应协议，当客户端向服务器发送一次请求时，首先是建立 TCP 连接，接着客户端向服务器发送请求，服务器处理完请求后会断开 TCP 连接，而在处理完客户端的请求后服务器不会存储客户端的任何信息，也就是说正常情况下，不同的客户端发送的请求对于服务器来说是一样的、独立的。那么，由于无法掌握客户端的信息，服务器就很难对不同客户做出不同的响应，包括权限控制、历史记录等。如何能够使服务器跟踪同一个客户端发出的连续请求呢？为了解决这个问题，早期的 Web 开发者设计了很多用户识别技术，包括承载用户身份信息的 HTTP 首部、跟踪客户端 IP 地址等，但这些技术都有这样或那样的缺陷，而由网景公司开发的 Cookie 技术是目前识别用户、实现会话控制最好的方法，我们可以将 session 等技术当做是 Cookie 技术的扩展。本章将简单介绍 Cookie 技术和 session 技术，然后详细介绍 Laravel 框架设计的 session 机制。在 Laravel 框架中没有使用 PHP 本身的 session 机制，因为 Laravel 框架设计者认为 PHP 本身的 session 机制很“丑陋”。

12.1 Cookie 技术

Cookie 一般可分为两类，即会话 Cookie 和持久 Cookie。会话 Cookie 一般用于临时应用，当用户退出浏览器后，Cookie 就会被删除；持久 Cookie 生存时间更长，即使退出浏览器、计算机重启依然存在。持久 Cookie 一般需要通过 Expires 或 Max-Age 参数进行特殊设置。

那么 Cookie 是如何工作的呢？通俗地讲，Cookie 相当于服务器端给客户端发送的用于给每个客户端进行标记的标识，而客户端在接收服务器添加标识的响应首部时会记录这个标识并在访问某个范围的服务器（可能就是这台服务器，也可能是一个域中所有的服务器，这跟服务器发送 Cookie 时的设置有关）时携带这个标识，当客户端再次访问这台服务器时，需要在请求头中附加这个标识，这样服务器就可以实现对客户端的跟踪了。

Cookie 在服务器和客户端之间是以什么形式传递的呢？实际上，是通过首部信息发送的，其中，Cookie 是以“名字 = 值”的形式构成的列表，服务器通过 Set-Cookie 或 Set-

Cookie2 响应首部将其发送给客户端，而客户端是通过 Cookie 请求首部发送给服务器端。知道了 Cookie 在服务器和客户端之间传递的方式，那么如何控制 Cookie 在两者之间的发送呢？这里以 PHP 编程语言为例，在服务器端实际上是通过 setcookie() 函数完成对 Cookie 响应首部的设置的，而客户端接收到这个响应首部时就会在本地存储相应的 Cookie 信息，当客户端访问相应的服务器时，就会查找对应服务器的 Cookie 信息并自动在请求首部中添加这些信息，而服务器需要通过 \$_COOKIE 全局数组来获取客户端发送的 Cookie 信息。下面通过一个简单的实例来讲解 Cookie 的机制。对应代码如下：

文件 index.php

```
<?php
setCookie("id","123456789");
echo "ok";
```

文件 index1.php

```
<?php
$id = $_COOKIE['id'];
echo $id;
```

这里我们首先访问 index.php 文件，通过 setcookie() 函数向客户端设置 Cookie，由于 Cookie 是通过报文首部传输的，所以在该函数前不允许有输出，即报文主体内容。下面是请求首部和响应首部内容。

Request Headers:

```
Accept: text/html,application/xhtml+xml,application/xml; q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8
Connection: keep-alive
Host: 127.0.0.1:10001
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36
```

Response Headers:

```
// 省略部分响应首部字段信息
Set-Cookie: id=123456789
Transfer-Encoding: chunked
X-Powered-By: PHP/5.6.1
```

当第一次访问服务器时，由于客户端没有 Cookie 信息，所以不携带，当访问 index.php 文件后，通过 setcookie() 函数在响应首部字段设置了 Cookie 信息，响应首部通过“Set-Cookie: id=123456789”向客户端发送该 Cookie 信息。下面是访问另一个文件 (index1.php) 的请求首部字段信息和响应首部字段信息。

Request Headers:

```
// 省略部分首部字段信息
```

```
Cookie: id=123456789
```

```
Host: 127.0.0.1:10001
```

Response Headers:

```
Connection: Keep-Alive
```

```
Content-Type: text/html; charset=UTF-8
```

```
Date: Tue, 22 Sep 2015 01:22:10 GMT
```

```
Keep-Alive: timeout=5, max=100
```

```
Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i mod_fcgid/2.3.9
```

```
Transfer-Encoding: chunked
```

```
X-Powered-By: PHP/5.6.1
```

当客户端没有关闭浏览器并再次访问服务器的其他文件(index1.php)时,客户端会到特定的目录下查找 Cookie 文件,根据服务器的地址识别对应的 Cookie 文件,并将其中的 Cookie 信息添加到请求首部字段中完成发送。对于本实例,请求首部字段会通过“Cookie: id=123456789”向服务器发送 Cookie 信息。

上面介绍的是会话 Cookie,如果希望使用持久 Cookie,则需要在 setcookie() 函数中加入其他可选参数,如“setcookie("id","123456789",time()+24*60*60);”表示设置 Cookie 使用期限为 1 天,那么当第一次访问服务器时,响应首部为“Set-Cookie: id=123456789; expires=Wed, 23-Sep-2015 01:34:14 GMT; Max-Age=86400”,这里直接设置 Cookie 的有效时间,在这段时间内,无论重启浏览器还是计算机,当再次访问该服务器时,都会携带设置的 Cookie 信息。

如果想删除 Cookie 中的数据,也使用 setcookie() 函数,只需要将其时间设置为当前时间之前的时间,使得客户端存储的 Cookie 数据过期,客户端就会自动注销对应的参数。例如,想删除 Cookie 中 id 参数的值,可以使用“setcookie("id","",time()-10);”这里将第二个参数设置成多大值不重要,只要满足小于当前时间即可,客户端就会将 id 的 Cookie 数据注销掉。

12.2 session 技术

12.2.1 session 的工作机制

session 技术相当于 Cookie 技术的升级版,Cookie 的工作机制是将信息记录在客户端,而 session 技术是将信息记录在服务器端,服务器存储信息的方式有很多种,可以是文件、数据库和内存等,这里以文件存储的方式介绍 session 的工作步骤。

(1) 客户端第一次访问某服务器。

(2) 服务器通过 Cookie 发送 sessionID 给客户端,并在服务器建立一个与 sessionID 同

名的文件用于存储信息，而 sessionID 不能重复，即不同客户端的 sessionID 是不同的。

(3) 客户端再次访问服务器时会携带服务器发送给客户端的 sessionID。

(4) 服务器根据客户端发送的 sessionID 查找对应的文件，读取文件中的内容。

通过上面的步骤可以看出，session 的工作依赖 Cookie 的工作，当然不用 Cookie 也可以实现 sessionID 的传递，如 URL，但是用 Cookie 最方便。

PHP 本身的 session 是如何工作的呢？首先，在需要共享客户端信息的文件中通过 session_start() 函数开启 session，然后就可以向 \$_SESSION 全局数组中存入或读取数据，而 \$_SESSION 数组与其他数组不同的是，当向该数组中添加数据时，PHP 还会将其中的数据序列化写入 session 文件中，每次开启 session 时，PHP 会将 session 文件中的数据读取到该全局数组中，实现数据共享的功能。下面以一个简单的例子来介绍 PHP 中 session 的工作机制。

文件 session1.php

```
<?php
// 开启 session，前面依然不能有输出
// 生成一个 sessionID 添加到响应首部，生成一个同名的文件存储在服务器中
session_start();
// 使用 $_SESSION 数组中添加内容，PHP 会将这个数组中的内容存储到文件中
$_SESSION['name'] = "xiaofang";
$_SESSION['age'] = 20;
```

文件 session2.php

```
<?php
// 判断客户端是否发送 sessionID，如果发送，则不再向客户端发送 sessionID
// 查找 sessionID 同名的文件，并将存储的数据读取到 $_SESSION 数组中
session_start();
print_r($_SESSION);
```

当客户端访问 session1.php 文件时，对应的请求首部代码如下所示，其中不包含 Cookie 信息，而服务器的响应报文则包含 sessionID，即 “Set-Cookie: PHPSESSID=pqfhplmkf1b372sr4cefi92797; path=/"，通过这个首部向客户端存储了 sessionID。

Request Headers:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8
Connection: keep-alive
Host: 127.0.0.1:10001
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36
```

Response Headers:

// 省略部分响应首部字段信息

Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i mod_fcgid/2.3.9

Set-Cookie: PHPSESSID=pqfhplmkf1b372sr4cefi92797; path=/
PHPSESSID=pqfhplmkf1b372sr4cefi92797

Transfer-Encoding: chunked

X-Powered-By: PHP/5.6.1

当客户端访问 session2.php 文件时, 请求首部将会包含 sessionID, 即 “Cookie: PHPSESSID=pqfhplmkf1b372sr4cefi92797”, 而 session2.php 文件同样通过 session_start() 函数开启 session, 但此时因为有请求发送的 sessionID 信息, 所以将不再发送 sessionID 的响应首部, 也不再重新创建 session 文件, 只是将服务器中与 sessionID 同名的文件中的数据读取到 \$_SESSION 数组中, 以便信息共享。

Request Headers:

Accept: text/html,application/xhtml+xml,application/xml;

q=0.9,image/webp,*/*;q=0.8

Accept-Encoding: gzip, deflate, sdch

Accept-Language: zh-CN,zh;q=0.8

Connection: keep-alive

Cookie: PHPSESSID=pqfhplmkf1b372sr4cefi92797

Host: 127.0.0.1:10001

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36

Response Headers:

Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0

Connection: Keep-Alive

Content-Type: text/html; charset=UTF-8

Date: Tue, 22 Sep 2015 11:54:11 GMT

Expires: Thu, 19 Nov 1981 08:52:00 GMT

Keep-Alive: timeout=5, max=100

Pragma: no-cache

Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i mod_fcgid/2.3.9

Transfer-Encoding: chunked

X-Powered-By: PHP/5.6.1

Session 中数据的删除相对于 Cookie 略微烦琐一些, 共需要四个步骤, 分别是开启 session、清空 \$_SESSION 数组、删除客户端 Cookie 中存储的 sessionID 和删除服务器中的 session 文件。当然, 如果只需要删除 session 中的部分数据, 那么只需要前两步就可以了。删除 session 数据的实例代码如下:

文件 session3.php

```
<?php
// 开启会话
session_start();
// 删除数组中的数据
// 删除单个数据
unset($_SESSION['name']);
// 清空整个数组中的内容
$_SESSION = array();
// 删除客户端 Cookie 中的 sessionID
setCookie(session_name(), "", time()-10, "/");
// 删除服务器中的 session 文件
session_destroy();
```

当客户端访问 session3.php 文件时，请求依然通过首部发送 sessionID，而服务器清空 \$_SESSION 数组后发送删除客户端 Cookie 中的 sessionID 首部，并销毁服务器中的 session 文件。

Request Headers:

```
// 省略部分请求首部字段
Connection: keep-alive
Cookie: PHPSESSID=pqfhplmkf1b372sr4cefi92797
Host: 127.0.0.1:10001
```

Response Headers:

```
// 省略部分响应首部字段
Pragma: no-cache
Server: Apache/2.4.10 (Win32) OpenSSL/1.0.1i mod_fcgid/2.3.9
Set-Cookie: PHPSESSID=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT;
Max-Age=0; path=/
X-Powered-By: PHP/5.6.1
```

12.2.2 session 的配置

在 PHP 中有很多关于 session 的配置参数，用于指定 session 的工作模式，如 session 文件的存储位置、session 的存储方式、Cookie 参数的设置等。如表 12.1 所示列出了 session 常用的配置项及其含义。

表 12.1 session 配置项及其含义

配置项	默认设置值	含 义
session.auto_start	0	0 表示不自动开启 session，如果设置为 1，则文件不需要使用 session_start() 函数就会自动开启 session，但对于面向对象编程，一般不建议开启

续表

配置项	默认设置值	含 义
session.Cookie_lifetime	0	0 表示客户端存储 sessionID 的 Cookie 为会话 Cookie，即当浏览器关闭时，Cookie 的值注销
session.Cookie_path	/	表示设置客户端 Cookie 的路径为根目录
session.name	PHPSESSID	设置 Cookie 的名称
session.save_handler	files	表示 session 数据的存储方式，以文件方式存储
session.save_path	/temp	表示 session 的存储位置，该位置是相对于网站根目录的
session.gc_maxlifetime	1440	表示 session 文件的过期时间，单位为秒，当 session 文件超过这个时间时，服务器的 session 垃圾回收器会将其删除
session.gc_divisor	1000	与 session.gc_probability 共同使用
session.gc_probability	1	与 session.gc_divisor 共同使用，表示用户操作 1000（session.gc_divisor）次 session 会有 1（session.gc_probability）次机会启动垃圾回收处理

在通过 PHP 的配置文件设置 session 工作模式的过程中，主要有三个方面内容：一是对 Cookie 的设置，包括设置 Cookie 的名称、生存时间等；二是服务器对 session 存储的设置，包括存储的位置和方式等；三是对过期 session 文件的回收机制，包括 session 文件的过期时间和 session 文件回收启动的频率等。通过上面三个部分的设置，PHP 自身的 session 机制已经可以正常工作了。

PHP 自身提供的 session 机制虽然功能比较完善，但是却有很多缺点，比如存储方式以文件形式存储，如果用户很多，会同时产生大量的 session 文件，而通过名称查找对应的 session 文件效率很低。另外，session 文件中的内容是以明文进行存储的，无法实现自动的加密处理等。这些缺点使得在一些大型 Web 开发时需要对 session 进行自定义设计，实现 session 机制的定制化，如将 session 内容存在数据库中、对 session 储存内容进行自动加密和解密功能等。这里对 session 的自定义设计就不进行详细讲解了，因为这种自定义设计依然是以原来的形式使用，只是重新设计了 session 的开启、关闭、读取、写入和销毁等函数，而 Laravel 框架完全重新设计了 session 的处理机制，包括它的使用方式。下面将介绍 Laravel 框架中的 session 机制。

12.3 Laravel 框架中的 session 机制

上文中提到，Laravel 框架重新设计了 session 的处理机制，那么 Laravel 框架中有哪些 session 的关键技术及如何使用这个全新的 session 呢？首先介绍 session 整个工作过程的步骤（这里依然以 Cookie 作为 session 的驱动，即 sessionID 是通过 Cookie 传输的）。

（1）当客户端访问服务器时，服务器将开启 session，检测请求的 Cookie 中是否携带

sessionID，如果携带则使用该 sessionID，如果没有则新产生一个 sessionID。这个过程可以称为 session 的启动阶段。

(2) 根据 sessionID 来恢复之前存储的数据，在请求处理期间可以使用恢复的数据，同时也可以向 session 中继续添加或删除数据。这个过程可以称为 session 的操作阶段。

(3) 当返回响应时，将 session 中的数据存储在相应的位置，以备下一次请求到来时使用并发送 sessionID 的 Cookie。这个过程可以称为 session 的关闭阶段。

下面分三个阶段介绍 Laravel 框架中 session 的工作机制。

12.3.1 session 的启动

在本书第 7 章中介绍了请求首先经过中间件的处理，在初始情况下，Laravel 框架共包含验证维护模式、Cookie 加密、添加响应 Cookie、开启会话、共享 session 错误和 CSRF 保护六个中间件，而 session 的启动就是在开启会话中间件中完成的。通过中间件处理请求就是通过中间件实例的 handle() 函数完成的，对这部分流程不清楚的读者可以查看第 7 章中的中间件部分。那么对 session 启动的介绍就从开启会话中间件的 handle() 函数讲起，下面是该函数的代码。

文件 Illuminate\Session\Middleware\StartSession.php

// 处理一个输入请求

```
public function handle($request, Closure $next)
{
    $this->sessionHandled = true;
    if ($this->sessionConfigured()) {
        $session = $this->startSession($request);
        $request->setSession($session);
    }
    $response = $next($request);
    if ($this->sessionConfigured()) {
        $this->storeCurrentUrl($request, $session);
        $this->collectGarbage($session);
        $this->addCookieToResponse($response, $session);
    }
    return $response;
}
```

一个 handle() 函数就包含了 session 开启和关闭的主要流程，其中，“\$response = \$next(\$request);” 语句之前的代码为 session 开启阶段，它之后的代码为 session 关闭阶段，而这一句代码是以管道的形式将请求实例向下传递的。在开启阶段通过 sessionConfigured() 函数检测 session 驱动的配置，如果配置了 session 的驱动，则通过 startSession() 函数开启 session，并在请求中添加 session 实例对象。由于 Laravel 没有使用 PHP 的原生 session，所

以通过一个类的实例来管理 session 的内容。

实际上, session 的开启阶段可以分为四个步骤完成, 分别是检测配置、session 实例化、开启 session 和将 session 实例传递给请求实例。下面看看它们分别做了哪些工作。

1. 检测配置

Session 的配置是在文件 `laravel\config\session.php` 中完成的, 它主要配置了 session 的驱动(这里指的是以何种媒介存储)、生存时间、是否加密和 Cookie 名称等。下面给出 Laravel 检测驱动配置的代码实现过程。

文件 `Illuminate\Session\Middleware\StartSession.php`

```
// 确定 session 驱动是否配置
protected function sessionConfigured()
{
    return !is_null(Arr::get($this->manager->getSessionConfig(), 'driver'));
}

// 创建一个 session 中间件
public function __construct(SessionManager $manager)
{
    $this->manager = $manager;
}
```

文件 `Illuminate\Session\SessionServiceProvider.php`

```
// 注册 session 管理器实例
protected function registerSessionManager()
{
    $this->app->singleton('session', function ($app) {
        return new SessionManager($app);
    });
}
```

文件 `Illuminate\Session\SessionManager.php`

```
// 获取 session 配置
public function getSessionConfig()
{
    return $this->app['config']['session'];
}
```

开启 session 的过程用到了一个实例对象, 即 session 管理器 (SessionManager 类实例), 该实例对象是在 session 中间件 (StartSession 类) 初始化的过程中通过依赖注入生成的, 依赖注入的类为 `Illuminate\Session\SessionManager` 类, 该类在服务容器中通过核心别名的方式注册了别名 “session”, 而对于名称为 “session” 的服务是在请求处理准备环节的服务提供者注册过程中注册的, 即在 `SessionServiceProvider` 类中实现, 于是服务容器在解决依赖注入时

实例化的对象是名为“session”的服务，即 SessionManager 类实例。这是完成 session 中间件依赖注入的过程，虽然很绕，经过很多步骤，但 Laravel 框架中大部分依赖都是通过这个过程解决的，了解之后就掌握了 Laravel 框架的核心之一。

有了 session 管理器，接下来验证 session 的驱动配置。对 session 驱动配置的验证是通过 session 管理器实例获取 session 的配置信息（“\$this->app['config']['session']”），并检测其中是否存在“driver”项实现的，如果存在则进行下一步的开启工作。

2. session 实例化

Session 的开启其实就是完成 session 实例化的过程，在 Laravel 框架中，session 实例是 EncryptedStore 类或 Store 类的实例，前者是加密 session 实例，后者是非加密 session 实例，主要区别在于数据在存储和读取过程中是否对数据进行加密和解密，这里以 Store 类为例来介绍。Store 类是需要驱动的，这个驱动需要符合 SessionHandlerInterface 接口，这个接口定义了完成 session 功能的 7 个函数接口，即开启、关闭、读取、写入、销毁和回收，而驱动的形式可能不同，如文件驱动、数据库驱动和 Memcache 驱动，区别就是 session 数据存储的媒介不同，只要满足 session 驱动接口的类都可作为 Store 类的驱动。下面以文件驱动为例，看看 Laravel 框架是如何完成 session 实例化的，具体代码如下：

文件 Illuminate\Session\Middleware\StartSession.php

```
// 针对给定的请求实例开启 session
protected function startSession(Request $request)
{
    with ($session = $this->getSession($request)) -
>setRequestOnHandler($request);
    $session->start();
    return $session;
}

// 通过管理器获取 session 的实现，即实例对象
public function getSession(Request $request)
{
    $session = $this->manager->driver();
    $session->setId($request->Cookies->get($session->getName()));
    return $session;
}
```

文件 Illuminate\Support\Manager.php

```
// 获取驱动实例
public function driver($driver = null)
{
    $driver = $driver ?: $this->getDefaultDriver();
    if (!isset($this->drivers[$driver])) {
```

```

        $this->drivers[$driver] = $this->createDriver($driver);
    }
    return $this->drivers[$driver];
}

```

文件 Illuminate\Session\SessionManager.php

```

// 获取默认的 session 驱动名称
public function getDefaultDriver()
{
    return $this->app['config']['session.driver'];
}

```

文件 Illuminate\Support\Manager.php

```

// 创建一个新的驱动实例
protected function createDriver($driver)
{
    $method = 'create'.ucfirst($driver).'Driver';
    if (isset($this->customCreators[$driver])) {
        return $this->callCustomCreator($driver);
    } elseif (method_exists($this, $method)) {
        return $this->$method();
    }
    throw new InvalidArgumentException("Driver [$driver] not supported.");
}

```

Session 类实例化及配置是通过代码 “\$this->getSession(\$request)” 实现的，而这个实现过程可以分为三个步骤：第一步根据 session 配置信息通过 session 管理器获取负责 session 实例化函数的名称；第二步是调用该实例化函数完成 session 类的实例化；第三步是完成 sessionID 的获取。首先通过 getDefaultDriver() 函数获取 session 默认驱动的配置信息，即 “config/session.php” 文件中的 “driver” 项，默认 Laravel 框架设置为 “file”，然后通过 “\$method='create'.ucfirst(\$driver).'Driver';” 获取 session 实例化函数 “createFileDriver”，最后通过该函数完成 session 类的实例化。

文件 Illuminate\Session\SessionManager.php

```

// 创建一个文件 session 驱动实例
protected function createFileDriver()
{
    return $this->createNativeDriver();
}
protected function createNativeDriver()
{
    $path = $this->app['config']['session.files'];
    return $this->buildSession(new FileSessionHandler($this->app['files'],

```

```

$path));
    }
    // 构建 session 实例
    protected function buildSession($handler)
    {
        if ($this->app['config']['session.encrypt']) {
            return new EncryptedStore(
                $this->app['config']['session.Cookie'], $handler, $this->app['encrypter']
            );
        } else {
            return new Store($this->app['config']['session.Cookie'], $handler);
        }
    }
}

```

在 session 实例化过程中会通过配置信息判断 session 是否加密，默认情况下是不加密的，即 “encrypt’ => false”，于是会通过 buildSession() 函数完成 Store 类的实例化，而 Store 类即为 Laravel 框架 session 的管理类。Store 类的初始化需要两个重要参数，一个是 Cookie 名称，通过代码 “\$this->app[‘config’][‘session.Cookie’]” 获取，默认情况下该名称为 “Laravel_session”，另一个是驱动，即 session 数据在服务器中存储的形式，默认以文件形式存储，所以对应的驱动类为 FileSessionHandler。Cookie 的名称和驱动的类型都可以在配置文件中设置。

文件 Illuminate\Session\Store.php

```

// 设置 sessionID
public function setId($id)
{
    if (!$this->isValidId($id)) {
        $id = $this->generateSessionId();
    }
    $this->id = $id;
}

```

在完成 session 实例化后，还需要获取 sessionID，即检查请求的 Cookie 中是否有 sessionID（这里是通过 Cookie 名称来识别的，即 \$session->getName() 获取 Cookie 名称），如果有就在 session 实例的 \$id 属性中进行记录，否则通过 generateSessionId() 函数重新生成一个 sessionID。

3. 开启 session 和实例传递

完成 session 实例化后，接下来就是 session 的开启工作，是通过 “\$session->start();”（Store 类实例中的 start() 函数，这里以 Store 类为例）完成的，其实这一步就是根据 sessionID 将对

应的数据从相应的存储媒介中取出来，放到 Store 类实例的 \$attributes 属性数组中。下面给出部分源码：

文件 Illuminate\Session\Store.php

```
// 开启 session
public function start()
{
    $this->loadSession();
    if (!$this->has('_token')) {
        $this->regenerateToken();
    }
    return $this->started = true;
}

// 从驱动中加载 session 数据
protected function loadSession()
{
    $this->attributes = array_merge($this->attributes, $this->readFromHandler());
    foreach (array_merge($this->bags, [$this->metaBag]) as $bag) {
        $this->initializeLocalBag($bag);
        $bag->initialize($this->bagData[$bag->getStorageKey()]);
    }
}

// 从驱动中读取 session 数据
protected function readFromHandler()
{
    $data = $this->handler->read($this->getId());
    if ($data) {
        $data = @unserialize($this->prepareForUnserialize($data));
        if ($data !== false && $data !== null && is_array($data)) {
            return $data;
        }
    }
    return [];
}
```

Store 实例通过 readFromHandler() 函数完成数据的读取，而该函数是通过 “\$this->handler->read(\$this->getId())” 完成的，前面讲到了 session 的驱动必须满足 SessionHandlerInterface 接口，而该接口中定义了 read() 函数，所以无论 session 驱动是什么，只要满足这个接口就可以实现 session 的功能，后期可以根据情况实现自己的 session 驱动，只要满足这个接口，就能实现功能的扩展。所以，在编程过程中需要针对接口编程，而不是针对实现编程，这里就看到了针对接口编程的威力了，这也是 Laravel 框架易扩展的一个原因。

12.3.2 session 的操作

在完成了 session 的启动过程后，对 session 的操作就变得十分容易了。前面已经讲过，session 开启的最终结果是生成一个 session 实例 (Store 类实例，这里以非加密 session 为例，如果是加密 session，则是 EncryptedStore 类实例)。那么，在程序处理请求生成响应的过程中，就可以通过操作该实例来实现对 session 的操作。下面介绍如何获取 session 实例和 session 实例的操作两部分内容。

1. session 实例的获取

在开发 Laravel 框架程序的过程中，经常需要对 session 中的内容进行操作，用于跟踪记录客户端的信息。要操作 session 中的内容，首先需要获取 session 实例，session 实例的获取其实和请求实例的获取大致相同，主要有三种方法。

(1) 通过请求实例获取。

```
$session = $request->session();
```

由于在 session 的启动过程中，最后一步就是将 session 实例传递给请求实例，因此可以通过请求实例的 session() 方法直接获取 session 的实例对象。

(2) 通过外观形式 (facade) 调用相关函数。

```
session::put('name', 'xiaofang');
```

通过 session 外观类 (Illuminate\Support\Facades\Session 类，在外观别名中定义) 的 getFacadeAccessor() 方法返回的服务名称是 “session”，而对应该名称的实例对象是 SessionManager 类实例，所以会调用该类的 put() 方法，但该类没有定义这个方法，该类继承自 Manager 类，Manager 类定义 __call() 魔术方法对未定义的函数进行处理，于是会调用 \$this->driver() 的 put() 方法，而 \$this->driver() 返回正式记录的 session 实例即 Store 类实例，因此会调用 Store 类实例的方法。这是通过外观类获取 session 实例的过程，与请求类实例的获取方法类似。

(3) 通过辅助函数 session()。

```
$session = session()->driver();
```

通过辅助函数 session() 返回的其实是名称为 “session” 的服务，即 SessionManager 类实例，可以通过该实例的 driver() 方法获取当前配置的 session 实例。

2. session 实例的操作

对 session 实例的操作主要有数据存储、数据读取和数据删除，同时 Laravel 框架中的 session 相对于 PHP 本身的 session 来讲还增加了数据暂存的功能，下面分别进行讲解。

(1) 数据存储。

```
Session::put('name', 'xiaofang');
```

```
Session::push('contact.phone', '15566668888');
```

put() 函数将数据存储到 Store 类实例的 \$attributes 属性中，即增加了 “\$attributes[name]='xiaofang’”，而 push() 函数是以数组的形式保存到 \$attributes 属性中，即增加了 “\$attributes[contact][phone]=array('155566668888’)”。

(2) 数据读取。

```
$value = Session::get('name');
$value = Session::get('key', 'default');
$value = Session::pull('name');
$data = Session::all();
```

get() 函数获取 Store 类实例的 \$attributes 属性中对应键的值，如果提供两个参数，则第二个参数是在 session 无法获取对应值时返回的默认值；pull() 函数是在取回对应值时进行删除，也可以添加第二个参数作为默认返回值；all() 函数是返回 session 所有的值。

(3) 数据删除。

```
Session::forget('key');
Session::flush();
```

forget() 函数是删除 session 中对应的键的值；flush() 函数是清空 session 中所有的值。

(4) 数据暂存。

```
Session::flash('key', 'value');
Session::reflash();
Session::keep(['username', 'email']);
```

有些 session 中的数据想保留到下一次请求，当本次请求处理结束时，保存的数据删除，就可以使用数据暂存。flash() 函数是将数据暂存到 session 中；reflash() 函数是将数据刷新一次，使得本次应该删除的数据不再删除，而是等到下一次请求结束时再删除；keep() 函数是刷新指定的数据。

其实在 session 中，有一个键值为 flash，该键值用来存储暂存数据，而该键值下又有两个键值，分别是 old 和 new，用来记录暂存数据的新旧，即本次请求存储的暂存数据为新，上一次存储的暂存数据为旧，当本次请求结束时，会将旧的数据删除，而新的数据保留，到下一次请求时，上次新的暂存数据就变成了旧的数据，以此循环。所以，在 Laravel 框架中应该避免使用 flash 作为键值。

12.3.3 session 的关闭

Session 的关闭可以分为四个步骤：存储当前的 URL、垃圾回收、添加响应首部和存储 session 数据。下面给出部分源码来分析。

文件 Illuminate\Session\Middleware\StartSession.php


```

public function handle($request, Closure $next)
{
    // 省略 session 启动阶段部分代码
    $response = $next($request);
    if ($this->sessionConfigured()) {
        $this->storeCurrentUrl($request, $session);
        $this->collectGarbage($session);
        $this->addCookieToResponse($response, $session);
    }
    return $response;
}

// 如果需要存储当前的 URL
protected function storeCurrentUrl(Request $request, $session)
{
    if ($request->method() === 'GET' && $request->route() && !$request->ajax()) {
        $session->setPreviousUrl($request->fullUrl());
    }
}

// 进行垃圾回收
protected function collectGarbage(SessionInterface $session)
{
    $config = $this->manager->getSessionConfig();
    // 对任何请求都以彩票的形式来触发垃圾回收，如果触发了，就会回收过期的 session
    if ($this->configHitsLottery($config)) {
        $session->getHandler()->gc($this->getSessionLifetimeInSeconds());
    }
}

// 根据配置项判断是否回收垃圾
protected function configHitsLottery(array $config)
{
    return mt_rand(1, $config['lottery'][1]) <= $config['lottery'][0];
}

// 添加 session 的 Cookie 到响应实例中
protected function addCookieToResponse(Response $response, SessionInterface $session)
{
    if ($this->usingCookieSessions()) {
        $this->manager->driver()->save();
    }

    if ($this->sessionIsPersistent($config = $this->manager->getSessionConfig())) {
        $response->headers->setCookie(new Cookie(
            $session->getName(), $session->getId(), $this->

```

```

>getCookieExpirationDate(),
    $config['path'], $config['domain'], Arr::get($config, 'secure',
false)
    ));
}
}
// 通过配置项判断 session 的驱动是否是持久化的
protected function sessionIsPersistent(array $config = null)
{
    $config = $config ?: $this->manager->getSessionConfig();
    return ! in_array($config['driver'], array(null, 'array'));
}
// 在请求的周期中执行最后的操作
public function terminate($request, $response)
{
    if ($this->sessionHandled && $this->sessionConfigured() && !$this-
>usingCookieSessions()) {
        $this->manager->driver()->save();
    }
}
}

```

下面对上述过程做简单介绍。当前 URL 的存储需要满足请求方法为 get、具有路由并且非 Ajax 的请求。垃圾回收是通过“`$this->configHitsLottery($config)`”进行判断的，实际上是在配置文件的“lottery”项中进行配置的，该项为一个数组，第一个值为阈值，第二个值为产生随机数的最大值，通过 `mt_rand()` 函数生成 1 到“`$config['lottery'][1]`”之间的随机数（默认这个值为 100，即生成一个 1 到 100 之间的随机数），如果这个值小于“`$config['lottery'][0]`”（默认值为 2），则触发垃圾回收，对应的几率为 1%，所以在一百次请求中可能会触发一次垃圾回收。添加 Cookie 到响应首部，首先判断 session 是否是持久存储，如果 session 的驱动存储不是 `array(null, 'array')` 数组中的值，就认为是持久化 session，默认情况下，session 的驱动存储为“file”，所以为持久存储。在持久存储情况下，会将 sessionID 设置到响应首部，即在响应实例的首部属性 `$headers` 中添加一个 Cookie 实例对象，用于发送 sessionID。session 数据的存储会在添加 Cookie 数据前判断 session 是否是基于 Cookie 的，如果是就在设置 Cookie 之前存储，否则就在程序关闭阶段调用中间的 `terminate()` 函数，`StartSession` 中间件的 `terminate()` 函数就是完成 session 存储的。

本章首先介绍了 Cookie 技术和 session 技术，并对 PHP 原生的这两种技术的用法进行了介绍。Laravel 框架重写了 session 实现，本章对 session 的启动、操作到关闭进行了详细的介绍，这里只是对以 file 存储的非加密 session 进行了介绍，其他类型虽然底层会有区别，但是流程是相同的，读者如果需要可以在了解本章的前提下摸索一下其他方式是如何实现的，也可以根据自身需要设计底层的实现方式，只要满足对应的接口就可以对 Laravel 框架进行扩展了。

第 13 章

消息队列

消息队列对于大型的 Web 项目来说是必不可少的一个模块，通过消息队列可以解决大并发和多种语言通信接口等问题。对于大并发的的问题，可以将耗时的任务或者不能同时大量并行的任务封装起来传输到消息队列中，由处理程序不断从消息队列中提取消息并进行处理，这样通过消息队列的缓冲可以使得在大并发情况下不再阻塞，如果性能不够用还可以添加多个处理任务从消息队列中获取消息进行处理。比如数据库的操作，当对数据库的读、写操作过多时就会存在锁表等问题，读的问题可以通过缓存等方案解决，写的问题就需要消息队列来解决。而且，在大型的 Web 项目开发中，很多情况下不可能通过一种语言实现，需要发挥不同语言的优势，比如 PHP，虽然在理论意义上它可以做 Web 开发中的所有事情，但是有些问题用它解决效率将会非常低，比如实时 socket 连接和分布式事务处理等。对于实时 socket 连接和推送等问题，node.js 更为擅长，实现效率也最高。对于分布式事务处理，Java 更为擅长，特别是与银行等金融行业的接口几乎都需要用 Java 实现。对于不同语言间的通信可以用消息队列来解决，PHP 接收用户的请求并把任务封装成消息投送到消息队列中，Java 或者 node.js 等语言编写的程序从消息队列中获取消息并进行下一步的任务处理。对于大型项目，PHP 可能只用来做与用户间的交互，而后期的处理由其他语言来处理。通过消息队列可以将多种语言、多台服务器连接起来，最终实现分布式开发。

Web 程序消息队列的一般实现如图 13.1 所示，用户请求触发消息生成，对于不需要实时处理并且耗时的的工作以消息的形式封装并存储到消息队列中，消息处理程序不断从消息队列中获取消息并进一步处理，使得请求和处理异步完成，从而实现程序的分布式处理。

在 Laravel 框架的手册中将消息队列更笼统地称为队列（queue），其实称为消息队列更加贴切，所以这里将延续这种叫法。Laravel 框架为大型项目分布式开发提供了完善的模块支持，其中消息队列提供了多种不同类型的驱动，包括数据库、Beanstalkd、IronMQ、Amazon SQS、redis、同步和 NULL。其中，NULL 类型用于简单的丢弃队列任务；同步类型用于本地直接使用，即发送任务和处理任务都在本地同时执行；数据库类型将任务消息存储在数据库中；redis 类型将任务消息存储在 redis 中；Beanstalkd、IronMQ 和 Amazon SQS 类型也存储在对应的系统中，相关的配置在文件 `Laravel/config/queue.php` 中。

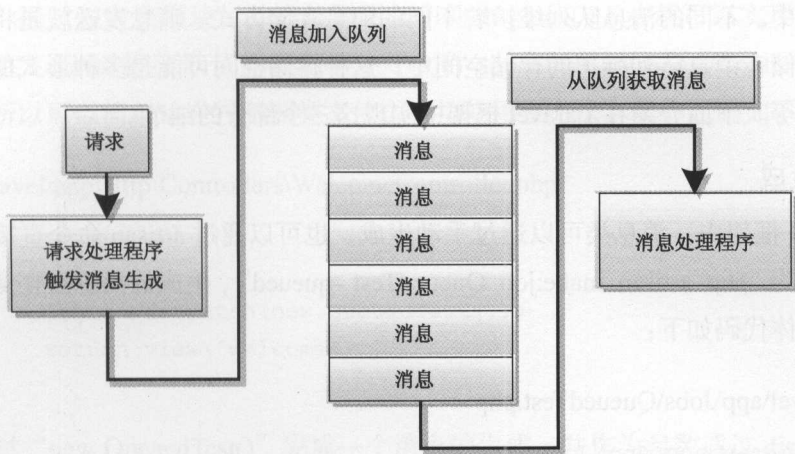


图 13.1 Web 程序消息队列的实现

为了了解 Laravel 框架中消息队列的实现细节，下面介绍同步类型和数据库类型两种驱动，其中同步类型是在本地同时完成消息的发送、提取和处理，可以用来了解消息队列的生命周期；数据库类型将消息的发送和处理分为两个不同的部分，可以用来了解消息队列的思想本质。

13.1 同步类型消息队列

在 Laravel 框架中，默认情况下设置的消息队列即为同步类型的消息队列，可以在配置文件中查看到消息队列的相关设置，其中配置文件为 `Laravel/config/queue.php`。默认的缺省设置为 `"default" => env('QUEUE_DRIVER', 'sync')`，其中 `QUEUE_DRIVER` 为在 `"Laravel/.env"` 中配置的主配置文件，该文件通过 `"QUEUE_DRIVER=sync"` 将消息队列的驱动设置为同步类型，如果想要修改消息队列的类型，直接修改 `.env` 文件中的 `QUEUE_DRIVER` 项就可以实现。下面将同步类型消息队列分为消息发送和消息处理两部分进行介绍。

13.1.1 消息发送

消息的发送需要实现消息生成、消息队列建立及消息的发送。在 Laravel 中，消息被称为工作（在后面将使用消息这样的叫法），是通过一个类来进行封装的，该类需要继承 `App\Jobs\Job` 类并存储在 `app/Jobs` 文件夹下，其中一个类代表一种消息类型，不同的消息类型可以封装不同的数据和数据处理方法。消息队列的建立是通过实例化相应的消息队列类来完成的，如同步类型的消息队列其实就是 `Illuminate\Queue\SyncQueue` 类的实例，而数据库类型的消息队列就是 `Illuminate\Queue\DatabaseQueue` 类的实例，不同消息队列类的实例管理不同的消息存储方式，如同步类型的消息队列直接将消息（也可称为工作）生成同步工作实例，即 `Illuminate\Queue\Jobs\Job` 类实例，而数据库类型的消息队列则是将消息序列化后

存储在数据库中。不同的消息队列维护着不同的消息存储方式。消息发送就是将封装并序列化后的消息存储在消息队列维护的存储空间中，这种存储空间可能是多种形式的，如文件、内存和数据库等。下面介绍在 Laravel 框架中实现这三个部分的细节。

1. 消息生成

在 Laravel 框架中，消息类可以通过手动生成，也可以通过 artisan 命令直接生成，对应的 artisan 命令为“php artisan make:job QueuedTest --queued”，生成的消息类存储在 \app\Jobs 文件夹下，具体代码如下：

文件 laravel\app\Jobs\QueuedTest.php

```
<?php namespace App\Jobs;
use App\Jobs\Job;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;
class QueuedTest extends Job implements SelfHandling, ShouldQueue
{
    use InteractsWithQueue, SerializesModels;
    public function __construct()
    {
    }
    // 处理一个消息
    public function handle()
    {
        echo "queue test success";
    }
}
```

通过命令生成的消息类包含一个构造函数和一个消息处理函数 handle()，其中一个消息就是该类的一个实例，在实例中可以封装相应的数据及处理这些数据需要的相应功能。这里只是在消息处理函数 handle() 中添加了一条输出响应用于测试。

2. 消息队列生成

对于整个消息队列来讲，消息队列的生成最关键，因为消息队列的实现方式决定了消息的封装形式、存储方式、获取方式及相应的性能等，在开源项目中有很多关于消息队列的，有些功能全，有些速度快，但实现的思想是基本相同的。在 Laravel 中，消息队列的生成和发送实现起来非常简单，只要在需要消息发送的类中包含 Illuminate\Foundation\Bus\DispatchesJobs 模块即可，这个模块其实是一个 trait，相当于 Ruby 中的模块（module），主要用于单继承的面向对象语言实现代码的共享，相应的介绍可以参考本书 PHP 的重要性质

的相关内容。通过包含该模块，并直接调用其中的 `dispatch()` 方法就可以实现消息队列的生成和发送。在 Laravel 框架默认的控制台中（文件 `app/Http/Controllers/Controller.php`）就包含该模块，所以可以简单地在控制器类文件中直接使用该方法，具体代码如下：

文件 `laravel/app/Http/Controllers/WelcomeController.php`

```
public function index()
{
    $this->dispatch(new QueuedTest());
    return view('welcome');
}
```

这里通过 “`new QueuedTest()`” 完成一个消息的生成，并作为参数通过 `dispatch()` 函数传递到消息队列中，其中该函数完成了消息队列的建立和消息的发送。下面介绍 Laravel 框架中消息队列的实现细节。

文件 `Illuminate/Foundation/Bus/DispatchesJobs.php`

```
// 分发消息到适当的处理模块
protected function dispatch($job)
{
    return app('Illuminate\Contracts\Bus\Dispatcher')->dispatch($job);
}
```

文件 `Illuminate/Bus/BusServiceProvider.php`

```
// 注册分发器服务
public function register()
{
    $this->app->singleton('Illuminate\Bus\Dispatcher', function ($app) {
        return new Dispatcher($app, function () use ($app) {
            return $app['Illuminate\Contracts\Queue\Queue'];
        });
    });
    $this->app->alias(
        'Illuminate\Bus\Dispatcher', 'Illuminate\Contracts\Bus\Dispatcher'
    );
    $this->app->alias(
        'Illuminate\Bus\Dispatcher', 'Illuminate\Contracts\Bus\
QueueingDispatcher'
    );
}
```

这里服务容器中关于 “`Illuminate\Contracts\Bus\Dispatcher`” 的注册是在 `Illuminate/Bus/BusServiceProvider` 类中实现的，实际是生成 `Illuminate/Bus/Dispatcher` 类实例，可以看到 Laravel 框架消息队列中消息的分发是通过命令总线实现的。

文件 \Illuminate\Bus\Dispatcher.php

```
// 创建一个新的命令分发器实例
public function __construct(Container $container, Closure $queueResolver
= null)
{
    $this->container = $container;
    $this->queueResolver = $queueResolver;
    $this->pipeline = new Pipeline($container);
}
// 分发一个命令到适当的处理模块
public function dispatch($command, Closure $afterResolving = null)
{
    if ($this->queueResolver && $this->commandShouldBeQueued($command)) {
        return $this->dispatchToQueue($command);
    } else {
        return $this->dispatchNow($command, $afterResolving);
    }
}
// 分发命令到对应的消息队列中
public function dispatchToQueue($command)
{
    $queue = call_user_func($this->queueResolver);
    if (! $queue instanceof Queue) {
        throw new RuntimeException('Queue resolver did not return a Queue
implementation.');
```

上面的代码在实现了命令总线分发器的实例化后，调用 `dispatchToQueue()` 方法实现消息队列的实例化及消息的分发。其中，消息队列实例的生成是通过“`call_user_func($this->queueResolver)`”实现的，而属性“`queueResolver`”存储的是命令总线的服务提供者中提供的消息队列实现的回调函数，即“`return $app['Illuminate\Contracts\Queue\Queue']`”，其中“`Illuminate\Contracts\Queue\Queue`”在服务容器（`Illuminate\Foundation\Application` 类）实例中被注册为核心别名“`queue.connection`”（参看服务容器类中的 `registerCoreContainerAliases()` 方法），“`queue.connection`”的服务是在 `Illuminate\Queue\QueueServiceProvider` 服务提供者中注册的。通过上述一系列过程即可实现消息队列的实例化。消息队列实例化过程的详细代码如下：

文件 Illuminate\Queue\QueueServiceProvider.php

```
// 注册消息队列控制器
protected function registerManager()
{
    $this->app->singleton('queue', function ($app) {
        $manager = new QueueManager($app);
        $this->registerConnectors($manager);
        return $manager;
    });
    $this->app->singleton('queue.connection', function ($app) {
        return $app['queue']->connection();
    });
}

// 注册消息队列中控制器的连接器
public function registerConnectors($manager)
{
    foreach (['Null', 'Sync', 'Database', 'Beanstalkd', 'Redis', 'Sqs',
'Iron'] as $connector) {
        $this->{"register{$connector}Connector"}($manager);
    }
}

// 注册同步消息队列连接器
protected function registerSyncConnector($manager)
{
    $manager->addConnector('sync', function () {
        return new SyncConnector;
    });
}

// 注册数据库消息队列连接器
protected function registerDatabaseConnector($manager)
{
    $manager->addConnector('database', function () {
        return new DatabaseConnector($this->app['db']);
    });
}
```

文件 Illuminate\Queue\QueueManager.php

```
// 添加一个消息队列连接处理器
public function addConnector($driver, Closure $resolver)
{
    $this->connectors[$driver] = $resolver;
}
```

在上文中提到，消息队列实例是通过获取服务容器中名为“queue.connection”的服务，

在服务提供者中绑定该服务名称的回调函数实现为“`return $app['queue']->connection()`”。这里又分为两个步骤实现，一是消息队列控制器实例的获取，即获取服务容器中名为“queue”的服务，二是通过消息队列控制器的 `connection()` 函数获取与配置文件对应的消息队列实例。以上代码主要实现第一个步骤，即消息队列控制器实例化过程，也就是获取服务容器中名称为“queue”的服务。通过上述代码可以看到，该服务名称回调函数完成两个功能，一个是完成消息队列控制器的实例化（`$manager = new QueueManager($app)`），二是在消息队列中注册消息队列连接器，该连接器封装了相应消息队列实例的获取方式，如“new SyncConnector”为同步消息队列连接器，这些是通过上面代码中的 `registerConnectors()` 函数实现的，为了简化只提供了同步消息队列和数据库消息队列连接器的注册函数。接下来将完成获取消息队列实例的操作。

文件 `Illuminate\Queue\QueueManager.php`

```
// 获取一个消息队列连接器实例
public function connection($name = null)
{
    $name = $name ?: $this->getDefaultDriver();
    if (! isset($this->connections[$name])) {
        $this->connections[$name] = $this->resolve($name);
        $this->connections[$name]->setContainer($this->app);
        $this->connections[$name]->setEncrypter($this->app['encrypter']);
    }
    return $this->connections[$name];
}

// 获取默认消息队列连接的名字
public function getDefaultDriver()
{
    return $this->app['config']['queue.default'];
}

// 解决一个消息队列连接
protected function resolve($name)
{
    $config = $this->getConfig($name);
    return $this->getConnector($config['driver']->connect($config);
}

// 根据给定的驱动获取消息队列连接器
protected function getConnector($driver)
{
    if (isset($this->connectors[$driver])) {
        return call_user_func($this->connectors[$driver]);
    }
    throw new InvalidArgumentException("No connector for [$driver]");
}
```



```
// 建立一个消息队列连接
public function connect(array $config)
{
    return new SyncQueue;
}
```

上述消息队列实例生成的过程分为两个步骤，第一步是获取消息队列连接器实例，如果连接器实例已经存在，则直接通过该连接器获取消息队列实例，如果不存在，则通过在消息队列控制器中注册的回调函数（上文中通过 `registerConnectors()` 函数注册的）获取相应实例，对应代码即为 `resolve()` 函数中的 “`$this->getConnector($config['driver'])`” 部分；第二步是通过消息队列连接器获取消息队列连接，其实就是所说的消息队列实例，对应 `resolve()` 函数中的 “`connect($config)`” 部分。至此，消息队列实例就已经生成了，接下来就是使用该实例完成消息发送和处理的功能。

3. 消息封装与发送

消息队列实例中已经封装了消息封装、发送和接收处理的一整套接口，消息的封装是通过不同的 `Job` 类实现的，在 `Laravel` 框架中，消息通过 `Job` 类的实例封装后再通过序列化封装成 `json` 格式，然后将其发送出去。具体实现代码如下：

文件 `\Illuminate\Bus\Dispatcher.php`

```
// 推送一条命令到消息队列实例
protected function pushCommandToQueue($queue, $command)
{
    if (isset($command->queue) && isset($command->delay)) {
        return $queue->laterOn($command->queue, $command->delay,
            $command);
    }
    if (isset($command->queue)) {
        return $queue->pushOn($command->queue, $command);
    }
    if (isset($command->delay)) {
        return $queue->later($command->delay, $command);
    }
    return $queue->push($command);
}
```

文件 `Illuminate\Queue\SyncQueue.php`

```
// 推送一条新的消息到队列
public function push($job, $data = '', $queue = null)
{
    $queueJob = $this->resolveJob($this->createPayload($job, $data,
        $queue));
}
```

```

try {
    $queueJob->fire();
    $this->raiseAfterJobEvent($queueJob);
} catch (Exception $e) {
    $this->handleFailedJob($queueJob);
    throw $e;
} catch (Throwable $e) {
    $this->handleFailedJob($queueJob);
    throw $e;
}
return 0;
}

```

消息队列除了类型之外还有名称和延迟时间，名称和延迟时间可以在消息实例中进行设置，也可以通过相应的函数进行设置，这些是在 `App\Jobs\Job` 类中通过 `trait(Illuminate\Bus\Queueable)` 实现的。普通的消息实例可以直接通过消息队列实例的 `push()` 方法进行发送。`push()` 方法首先完成消息的再次封装，即通过 `createPayload()` 方法将消息实例序列化后生成 json 格式，然后将封装后的消息发送到消息队列。在一般的消息队列中，发送和处理是在不同模块中完成的，如在数据库消息队列中，`push()` 方法只是将消息存储到相应的数据库中；在 redis 消息队列中，`push()` 方法将消息存储到 redis 数据库或内存中。对于同步类型的消息队列，消息的发送和处理是同时完成的，即将封装的消息直接通过 `resolveJob()` 进行获取，并通过 `fire()` 函数进行处理。这里重点介绍消息封装、发送和处理的过程，在后面数据库消息队列中会介绍一般消息队列的使用方法。同步类型消息队列中消息的封装实现如下：

文件 `Illuminate\Queue\Queue.php`

```

// 通过给定的消息和数据创建一个载荷字符串
protected function createPayload($job, $data = '', $queue = null)
{
    if ($job instanceof Closure) {
        return json_encode($this->createClosurePayload($job, $data));
    } elseif (is_object($job)) {
        return json_encode([
            'job' => 'Illuminate\Queue\CallQueuedHandler@call',
            'data' => ['command' => serialize(clone $job)],
        ]);
    }
    return json_encode($this->createPlainPayload($job, $data));
}

```

文件 `Illuminate\Queue\SerializesModels.php`

```

// 为序列化准备实例
public function __sleep()

```

```

{
    $properties = (new ReflectionClass($this))->getProperties();
    foreach ($properties as $property) {
        $property->setValue($this, $this->getSerializedPropertyValue(
            $this->getPropertyValue($property)
        ));
    }
    return array_map(function ($p) { return $p->getName(); }, $properties);
}

```

在 Laravel 框架下，不同消息队列类型对于消息的封装是统一的，都是通过 `createPayload()` 函数实现的，对于消息实例，首先是消息的接口处理，即 “`job`” => “`Illuminate\Queue\CallQueuedHandler@call`”，然后是对消息实例的序列化，在序列化之前通过 `clone()` 函数复制消息实例，在复制前会调用魔术方法 `__sleep()` 实现属性的转换。

13.1.2 消息处理

在经过消息封装后，同步类型的消息队列没有消息存储和获取的过程，而是直接进入消息处理阶段，但是一般的消息都会有消息存储和获取阶段。消息的处理实现如下：

文件 `Illuminate\Queue\SyncQueue.php`

```

// 处理一个同步类型消息实例
protected function resolveJob($payload)
{
    return new SyncJob($this->container, $payload);
}

```

文件 `Illuminate\Queue\Jobs\SyncJob.php`

```

// 处理一个消息
public function fire()
{
    $this->resolveAndFire(json_decode($this->payload, true));
}

```

消息处理的第一步是将封装的消息 (`$payload`) 封装在 `SyncJob` 类中，该类提供了对于消息解码和分发处理的接口。通过 `json_decode()` 将 json 格式的数据进行解码，然后通过 `resolveAndFire()` 函数进行消息处理。下面是处理过程的代码：

文件 `Illuminate\Queue\Jobs\Job.php`

```

// 通过消息处理函数处理消息
protected function resolveAndFire(array $payload)
{
    list($class, $method) = $this->parseJob($payload['job']);
}

```



```

        $this->instance = $this->resolve($class);
        $this->instance->{$method}($this, $this->resolveQueueableEntities($payload['data']));
    }

```

文件 Illuminate\Queue\CallQueuedHandler.php

// 处理消息实例

```

public function call(Job $job, array $data)
{
    $command = $this->setJobInstanceIfNecessary(
        $job, unserialize($data['command'])
    );
    $this->dispatcher->dispatchNow($command, function ($handler) use ($job) {
        $this->setJobInstanceIfNecessary($job, $handler);
    });
    if (! $job->isDeletedOrReleased()) {
        $job->delete();
    }
}

```

文件 Illuminate\Bus\Dispatcher.php

// 在当前进程中将命令发送给适当的处理模块

```

public function dispatchNow($command, Closure $afterResolving = null)
{
    return $this->pipeline->send($command)->through($this->pipes)->then(function ($command) use ($afterResolving) {
        if ($command instanceof SelfHandling) {
            return $this->container->call([$command, 'handle']);
        }
        $handler = $this->resolveHandler($command);
        if ($afterResolving) {
            call_user_func($afterResolving, $handler);
        }
        return call_user_func(
            [$handler, $this->getHandlerMethod($command)], $command
        );
    });
}

```

在 Laravel 框架中，消息的处理统一通过 CallQueuedHandler 类中的 call() 函数实现，该函数会通过命令总线转发器实例的 dispatchNow() 方法来处理消息，然后调用消息实例中的 handle() 处理函数来处理，在本例中即为消息类 App\Jobs\QueuedTest 中的 handle() 函数。

以上就是 Laravel 框架中关于同步类型消息的整个执行流程。一般消息队列的执行流程

可以分为七个步骤，分别为消息实例生成（工作生成）、消息队列实例生成（队列连接生成）、消息序列化封装、消息存储（消息推送）、消息获取（消息抛出）、消息处理类封装和消息处理。其中，括号内是根据 Laravel 文档和源码注释的直译的叫法。需要进一步说明的是，在同步类型的消息队列中，消息序列化封装后直接就将其封装成消息处理类完成消息处理，中间没有消息存储和消息获取的过程。同时，由于整个消息队列的实现都是在 Laravel 框架下实现的，所以消息生成到处理的各个过程都是在 Laravel 框架下实现的，实际上在大型的分布式开发中一般是不会用一种语言实现的，Laravel 框架可能需要做的是消息的生成和发送，而消息的获取和处理是其他语言或模块完成的，如果是这样的软件架构就需要对 Laravel 框架的部分内容进行修改，好在我们对消息队列的整个流程细节都了解了，可以很容易地修改相应的流程结构。

13.2 数据库类型消息队列

在同步类型消息队列中，前面分析了 Laravel 框架关于消息生成和处理的整个流程，不过同步类型消息队列中缺少消息存储和消息获取两个步骤，下面通过数据库类型的消息队列来讲解这两个步骤，其他步骤和同步类型消息队列中的步骤基本相同，这里不再赘述。

13.2.1 参数配置

在 Laravel 中，默认情况下消息队列设置的是同步类型，如果需要设置成数据库类型，只需要将 .env 文件中的 QUEUE_DRIVER 配置项设置成 database 即可。由于数据库消息队列中的消息是存储在数据库中的，所以需要配置数据库中的相应选项，即 DB_HOST、DB_DATABASE、DB_USERNAME 和 DB_PASSWORD 四项，分别是数据库的 IP 地址、数据库名称、用户名和密码。对于本实例相应的设置如下：

```
DB_HOST=localhost
DB_DATABASE=Laravel5-1
DB_USERNAME=root
DB_PASSWORD=root
QUEUE_DRIVER=database
```

13.2.2 数据表的建立

由于需要数据库存储消息，所以要建立相应的数据表，在 Laravel 中提供了相应的 artisan 命令用于创建数据库消息队列的数据表迁移文件，相应的命令是“php artisan queue:table”。通过该命令创建的数据表存储在 laravel\database\migrations 目录下，相应的迁移文件如下：

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateJobsTable extends Migration
{
    // 执行迁移
    public function up()
    {
        Schema::create('jobs', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('queue');
            $table->longText('payload');
            $table->tinyInteger('attempts')->unsigned();
            $table->tinyInteger('reserved')->unsigned();
            $table->unsignedInteger('reserved_at')->nullable();
            $table->unsignedInteger('available_at');
            $table->unsignedInteger('created_at');
            $table->index(['queue', 'reserved', 'reserved_at']);
        });
    }
    // 返回迁移
    public function down()
    {
        Schema::drop('jobs');
    }
}
```

下面就可以通过 artisan 数据库迁移命令实现数据表的建立，命令为“php artisan migrate”，生成的数据表结构如图 13.2 所示。

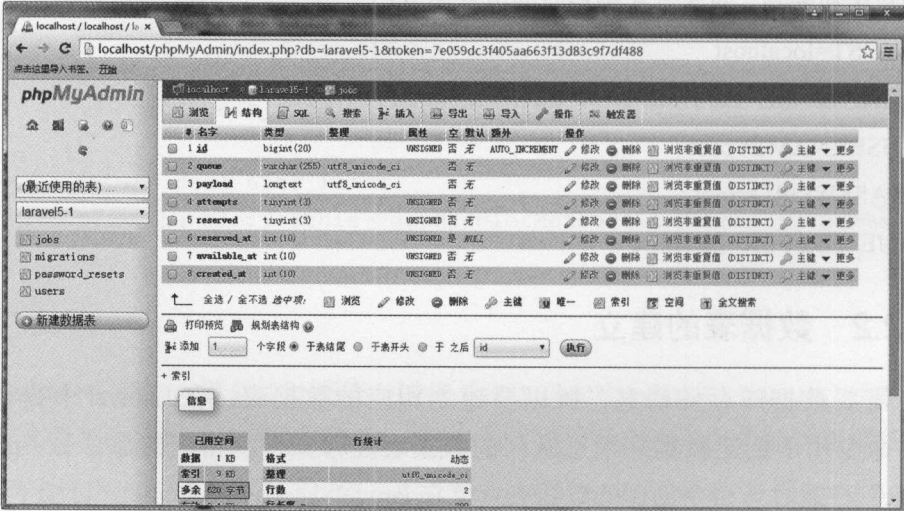


图 13.2 数据表结构

13.2.3 消息的生成、发送与处理

这里依然在控制器中调用 Illuminate\Foundation\Bus\DispatchesJobs 模块中的 dispatch() 方法实现消息的生成和发送。在数据库类型的消息队列中, dispatch() 方法只实现了消息实例生成、消息队列实例生成、消息序列化封装和消息存储四个步骤,对于消息获取、消息处理类封装和消息处理三个步骤虽然 Laravel 框架提供相应的接口,但需要读者自己设置处理过程。下面给出了在控制器中消息的发送和处理过程,具体代码如下:

文件 laravel\app\Http\Controllers\WelcomeController.php

```
public function index()
{
    // 消息的生成与发送
    $id = $this->dispatch(new QueuedTest());
    // 消息的获取与处理
    $queue = app('Illuminate\Contracts\Queue\Queue');
    $queueJob = $queue->pop();
    $queueJob->fire();
    return view('welcome');
}
```

13.2.4 消息存储

在 Laravel 框架中,不同类型的消息队列大部分步骤的接口实现都是相同的,下面介绍同步类型消息队列中不包含的消息存储过程。对于不同类型的消息队列,消息的发送都是通过相应类型消息队列实例(数据库类型消息队列为 Illuminate\Queue\DatabaseQueue 类的实例)的 push() 函数实现的,相应的实现代码如下:

文件 Illuminate\Queue\DatabaseQueue.php

```
// 推送一个新的消息到队列中
public function push($job, $data = '', $queue = null)
{
    return $this->pushToDatabase(0, $queue, $this->createPayload($job,
    $data));
}
// 根据给定的延迟时间推送一个原始的消息载荷到数据库
protected function pushToDatabase($delay, $queue, $payload, $attempts = 0)
{
    $attributes = $this->buildDatabaseRecord(
        $this->getQueue($queue), $payload, $this->getAvailableAt($delay),
    $attempts
    );
    return $this->database->table($this->table)->insertGetId($attributes);
}
```

```
}
// 根据给定的消息实例创建一个插入数据库的数组
protected function buildDatabaseRecord($queue, $payload, $availableAt,
$attempts = 0)
{
    return [
        'queue' => $queue,
        'payload' => $payload,
        'attempts' => $attempts,
        'reserved' => 0,
        'reserved_at' => null,
        'available_at' => $availableAt,
        'created_at' => $this->getTime(),
    ];
}
// 如果队列参数为空, 则获取默认值
protected function getQueue($queue)
{
    return $queue ?: $this->default;
}
```

这里 `push()` 函数是通过 `pushToDatabase()` 函数实现消息实例载荷插入数据库的, 在该函数中首先根据消息实例和设定的参数通过 `buildDatabaseRecord()` 生成插入数据库的数据数组, 然后通过属性 `$database` 来实现数据的插入, 该实例其实是数据库查询构造器的实例, 即 `$app['db']`, 最后返回的是插入数据表中的 ID。

13.2.5 消息获取

在数据库类型的消息队列中, 通过 `push()` 函数将消息存储在数据库中, 实际上对消息的处理可以使用任何方式, 只需要不断查询数据库中相应数据表中的内容, 如果有内容就说明有消息没有处理, 就可以提取消息进行处理, 完成后删除相应数据表中的内容。这里依然借助 Laravel 框架提供的相应接口进行消息的获取和处理。实现的代码在 13.2.3 小节已经给出了, 其中代码 “`$queue = app('Illuminate\Contracts\Queue\Queue');`” 用来获取数据库类型消息队列实例, 由于消息队列实例是在 `Illuminate\Queue\QueueManager` 类中的 `$connections` 属性中存储的, 所以获取的消息队列实例与消息发送的实例是同一个, 然后通过该实例的 `pop()` 函数实现消息的获取, 其实现的详细代码如下:

文件 `Illuminate\Queue\DatabaseQueue.php`

```
// 从队列中抛出下一个消息
public function pop($queue = null)
{
    // ...
}
```

```

    $queue = $this->getQueue($queue);
    if (! is_null($this->expire)) {
        $this->releaseJobsThatHaveBeenReservedTooLong($queue);
    }
    if ($job = $this->getNextAvailableJob($queue)) {
        $this->markJobAsReserved($job->id);
        $this->database->commit();
        return new DatabaseJob(
            $this->container, $this, $job, $queue
        );
    }
    $this->database->commit();
}
// 从队列中获取下一个可用的消息
protected function getNextAvailableJob($queue)
{
    $this->database->beginTransaction();
    $job = $this->database->table($this->table)
        ->lockForUpdate()
        ->where('queue', $this->getQueue($queue))
        ->where('reserved', 0)
        ->where('available_at', '<=', $this->getTime())
        ->orderBy('id', 'asc')
        ->first();
    return $job ? (object) $job : null;
}
// 将给定的消息 ID 标记为保留
protected function markJobAsReserved($id)
{
    $this->database->table($this->table)->where('id', $id)->update([
        'reserved' => 1, 'reserved_at' => $this->getTime(),
    ]);
}

```

文件 Illuminate\Queue\Jobs\DatabaseJob.php

```

// 处理消息
public function fire()
{
    $this->resolveAndFire(json_decode($this->job->payload, true));
}

```

这里通过消息队列实例中的 `pop()` 函数获取消息，该函数首先通过 `getNextAvailableJob()` 函数在数据表中获取下一个可用的消息，然后通过 `markJobAsReserved()` 函数将其标记为保留，并记录保留的时间，用于后期的消息删除或出现错误后的错误处理，获取的消息依然被

封装在 `Illuminate\Queue\Jobs\DatabaseJob` 类实例中，该类中同样提供了消息解码和处理的相应接口，于是通过该类中的 `fire()` 函数实现消息的处理，之后的过程与同步类型消息队列相似，这里就不再赘述了。

本章介绍了 Laravel 框架中消息队列的基本内容。消息队列是构建分布式系统不可或缺的一个组件，它提供了业务处理模块间的缓冲通信，使得各模块间处理过程从同步变为了异步，从而极大地增强了程序构建的灵活性和数据处理的自由度，在构建大型项目的过程中，要将可以异步工作的模块划分出来，通过消息队列进行连接通信，这种设计方式可使各模块间耦合度降低，即减少错误的产生，也为程序构建提供了细粒度的划分。

第 14 章

认证与数据验证

认证与数据验证在 Web 开发中扮演着重要的角色，在项目开发中经常需要对不同的用户分配不同的权利，如内容的修改、查看和添加等，如果认证和数据验证做得不好，那么软件的安全性和正常使用都无法保证。Laravel 框架提供了完整的认证和数据验证模块，使得开发相关的功能变得非常简单，下面以 Laravel 默认自带的相关模块来介绍认证和数据验证等功能的开发。

14.1 认证

14.1.1 认证模块的配置

对 Laravel 框架中不同的模块，可以通过配置文件从总体上了解该模块涉及的相关内容，其中模块配置文件都存储在 config 文件夹内，而认证模块的配置文件为 auth.php。下面是认证模块的主要配置内容。

文件 laravel\config\auth.php

```
return [
    'driver' => 'eloquent',
    'model' => App\User::class,
    'table' => 'users',
    'password' => [
        'email' => 'emails.password',
        'table' => 'password_resets',
        'expire' => 60,
    ],
];
```

配置文件的功能其实就是返回一个配置数组。在配置数组中，“driver”项用于设置默认的认证驱动，指定通过何种方式检索和认证存储用户信息的数据库，支持 database 和

eloquent 两种方式。当选择 eloquent 作为认证驱动时，将通过 Eloquent ORM 方式操作数据库，这时需要知道用哪个 eloquent 模型类来检索用户，可通过“model”配置项来设置，默认情况下使用 users 模型类，也可以使用任何需要的。当选择 database 作为认证驱动时，会通过查询构造器来操作数据库，这时需要知道用哪个表来检索用户，可通过“table”配置项来设置，默认情况下使用 users 表，也可以设置需要的其他表。“password”配置项设置的是关于重置密码相关的选项，包括密码重置视图的内容、重置密码时保存重置令牌的表名和令牌的有效时间，时间单位为分钟。Laravel 框架默认使用 eloquent 驱动来完成用户的认证功能。

14.1.2 数据表的建立

在了解了认证模块的配置后，需要创建用户认证和密码重置的数据表，在 Laravel 框架中，默认已经包含了相关数据表的数据迁移文件，存放在 database\migrations 文件夹下，可以通过 artisan 命令“php artisan migrate”执行数据表的创建，其中两个表的迁移代码如下：

文件 laravel\database\migrations\2014_10_12_000000_create_users_table.php

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password', 60);
        $table->rememberToken();
        $table->timestamps();
    });
}
```

文件 laravel\database\migrations\2014_10_12_100000_create_password_resets_table.php

```
public function up()
{
    Schema::create('password_resets', function (Blueprint $table) {
        $table->string('email')->index();
        $table->string('token')->index();
        $table->timestamp('created_at');
    });
}
```

这里通过数据迁移命令创建了两个表，分别是 users 表和 password_resets 表。users 表中包含一个 remember_token 字段，为非空的 100 个字符的字段，该字段用于保存用户认证的 session。两个表的结构分别如图 14.1 和图 14.2 所示。

#	名字	类型	整理	属性	空	默认	额外
<input type="checkbox"/>	1 id	int (10)		UNSIGNED	否	无	AUTO_INCREMENT
<input type="checkbox"/>	2 name	varchar (255)	utf8_unicode_ci		否	无	
<input type="checkbox"/>	3 email	varchar (255)	utf8_unicode_ci		否	无	
<input type="checkbox"/>	4 password	varchar (60)	utf8_unicode_ci		否	无	
<input type="checkbox"/>	5 remember_token	varchar (100)	utf8_unicode_ci		是	NULL	
<input type="checkbox"/>	6 created_at	timestamp			否	0000-00-00 00:00:00	
<input type="checkbox"/>	7 updated_at	timestamp			否	0000-00-00 00:00:00	

☐ 全选 / ☐ 全不选 ☐ 选中项: ☐ 浏览 ☐ 修改 ☐ 删除 ☐ 主键 ☐ 唯一 ☐ 索引

图 14.1 users 数据表结构

#	名字	类型	整理	属性	空	默认	额外
<input type="checkbox"/>	1 email	varchar (255)	utf8_unicode_ci		否	无	
<input type="checkbox"/>	2 token	varchar (255)	utf8_unicode_ci		否	无	
<input type="checkbox"/>	3 created_at	timestamp			否	0000-00-00 00:00:00	

图 14.2 password_resets 数据表结构

14.1.3 添加用户认证路由

Laravel 框架默认情况下包含两个与认证相关的控制器，分别是认证控制器（AuthController 类）和密码重置控制器（PasswordController 类），其中认证控制器用于注册和认证用户，而密码重置控制器用于忘记密码的用户重置密码。这两个控制器位于 App\Http\Controllers\Auth 目录下，并且默认已经包含了常用的方法，一般情况下可以直接使用。有了控制器，还要设置相应的路由，用来连接用户请求和控制器。Laravel 文档中给出了相应的路由配置，可以直接添加到路由文件 app/Http/routes.php 中。下面是认证路由、注册路由和密码重置路由的代码。

文件 laravel\app\Http/routes.php

```
// 认证路由
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');
// 注册路由
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
// 密码重置请求链接路由
Route::get('password/email', 'Auth>PasswordController@getEmail');
Route::post('password/email', 'Auth>PasswordController@postEmail');
```

```
// 密码重置路由
Route::get('password/reset/{token}', 'Auth\PasswordController@getReset');
Route::post('password/reset', 'Auth\PasswordController@postReset');
```

14.1.4 认证视图的创建

在完成路由配置后，还需要添加认证视图文件，用于进行用户的注册和登录认证。在 Laravel 5.0 中默认给出了用户认证的视图模板，包含在 laravel\resources\views\auth 目录下，但在 Laravel 5.1 中去掉了认证的视图模板，而是在文档中给出了认证模板需要包含的基本元素，即与用户表和密码重置表的字段相对应的结构，可以根据基本元素自行设计相应的模板。下面分别是注册、登录、密码重置请求发送和密码重置视图的基本元素代码及对应的视图，需要注意的是这里提供的视图文件是 Laravel 5.0 中的视图文件，在 Laravel 5.1 以上版本已经不再包含这些文件了，只是在文档中给出了视图文件的基本代码，不过两者的基本元素是相同的，只是没有相关的样式设置而已。下面首先是注册视图的代码。

文件 laravel\resources\views\auth\register.blade.php

```
<form method="POST" action="/auth/register">
    {!! csrf_field() !!}

    <div>Name<input type="text" name="name" value="{{ old('name') }}"></div>

    <div> Email<input type="email" name="email" value="{{ old('email') }}"></div>

    <div> Password <input type="password" name="password"></div>
    <div> Confirm Password<input type="password" name="password_confirmation"> </div>

    <div> <button type="submit">Register</button> </div>
</form>
```

如图 14.3 所示为用户注册页面视图。

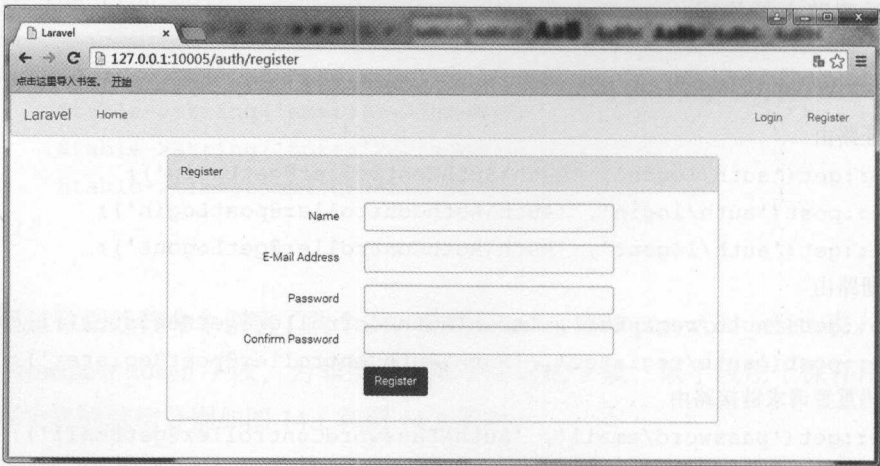


图 14.3 用户注册页面视图

下面是登录视图的代码。

文件 `laravel/resources/views/auth/login.blade.php`

```
<form method="POST" action="/auth/login">
    {!! csrf_field() !!}
    <div>Email<input type="email" name="email" value="{{ old('email')
}}"></div>
    <div>Password<input type="password" name="password" id="password"></
div>
    <div><input type="checkbox" name="remember"> Remember Me </div>
    <div> <button type="submit">Login</button></div>
</form>
```

如图 14.4 所示为用户登录页面视图。

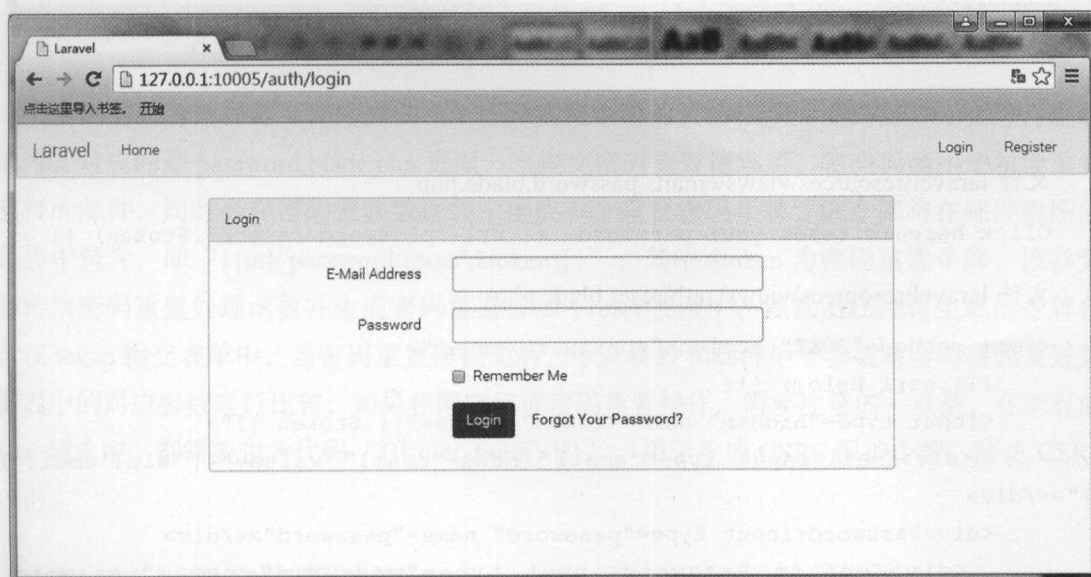


图 14.4 用户登录页面视图

下面是密码重置请求发送视图的代码。

文件 `laravel/resources/views/auth/password.blade.php`

```
<form method="POST" action="/password/email">
    {!! csrf_field() !!}
    <div>Email<input type="email" name="email" value="{{ old('email')
}}"></div>
    <div><button type="submit"> Send Password Reset Link</button></div>
</form>
```

如图 14.5 所示为密码重置请求页面视图。

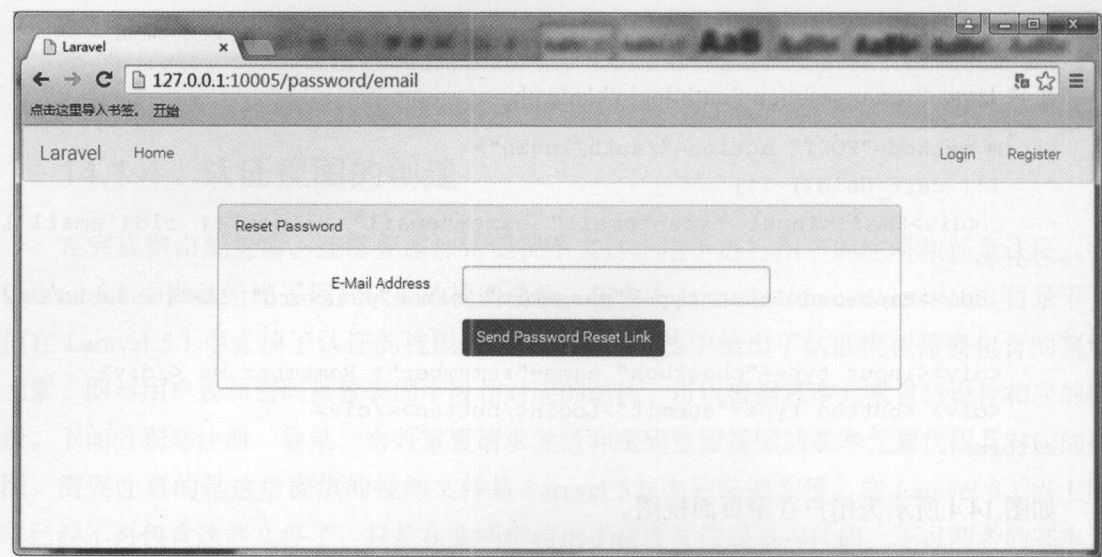


图 14.5 密码重置请求页面视图

下面是密码重置视图的代码。

文件 `laravel/resources/views/emails/password.blade.php`

Click here to reset your password: `{{ url('password/reset/'.$token) }}`

文件 `laravel/resources/views/auth/reset.blade.php`:

```
<form method="POST" action="/password/reset">
    {!! csrf_field() !!}
    <input type="hidden" name="token" value="{{ $token }}">
    <div>Email<input type="email" name="email" value="{{ old('email')
}}"></div>
    <div>Password<input type="password" name="password"></div>
    <div>Confirm Password<input type="password" name="password_
confirmation">    </div>
    <div><button type="submit">Reset Password</button></div>
</form>
```

如图 14.6 所示为密码重置页面视图。

上面的注册和登录视图比较容易理解，当注册或登录成功后会跳转到相应的页面，跳转的页面可以通过 `AuthController` 类中的 `$redirectPath` 属性进行设置，如“`protected $redirectPath = '/home';`”，如果登录不成功，则会再次跳转到登录页面，该跳转页面可以通过 `AuthController` 类中的 `$loginPath` 属性进行设置，如“`protected $loginPath = '/login';`”。

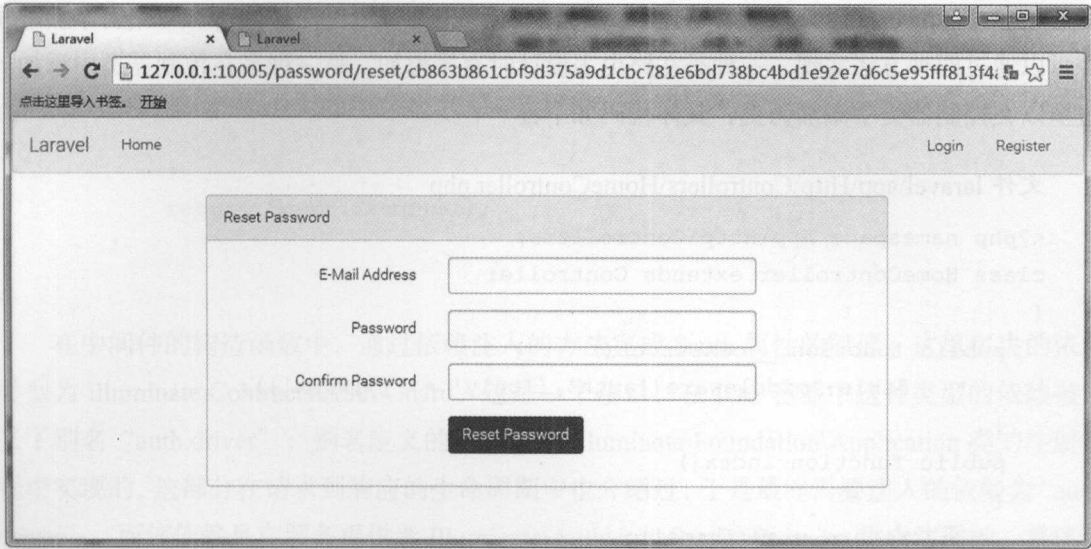


图 14.6 密码重置页面视图

密码重置的流程是，首先提供一个密码重置申请的页面，该页面只需要提交邮件地址即可，对应的是 `password.blade.php` 页面，当提交密码重置请求后，将会得到一个包含重置链接的邮件，同时会在密码重置数据表中生成一个重置密码令牌，该令牌将在邮件视图的链接中包含，即 “`{{url('password/reset/'.$token)}}`”，其中 `$token` 为密码重置令牌，该参数会传给密码重置处理函数并生成密码重置视图（reset 视图），该视图已经将生成的令牌包含在 token 提交表单中，当密码重置被提交后，令牌参数和邮件地址参数将会与密码重置数据表中的对应参数进行比较，如果相同则完成密码重置操作。需要注意的一点是，在所有的 post 请求中，都需要包含代码 “ `{!! csrf_field() !!}` ”，用于生成 CSRF 保护令牌，防止 CSRF 攻击。

14.1.5 用户权限认证

用户权限认证的根本目的就是对于特定的路由只允许通过认证的用户访问。在用户权限认证中一般可以通过三种方式完成，第一种是通过 Laravel 自带的路由中间件来对用户认证进行判断，用以对需要经过认证后用户访问的路由进行保护；第二种是获取经过认证的用户信息，如果可以获取到，则说明该用户通过认证；第三种是通过用户认证的方法判断用户是否已经通过认证。虽然用户认证的形式不同，但是底层都是使用 `Illuminate\Auth\AuthManager` 类来实现用户认证的，虽然该类提供了实现用户权限认证的方法，但却是通过魔术方法 `__call()` 来实现的，而实际的操作是通过底层的 `Illuminate\Auth\Guard` 类来实现的。下面详细介绍其中一种实现的代码流程，其他方法最终也是通过这些类来实现的。

1. 通过路由中间件进行用户权限认证

Laravel 框架默认自带了一个用户权限认证的路由中间件，当某些路由需要通过权限认

证的用户才能够访问时，只需要在这些路由上加入该中间件进行保护即可。对路由添加中间件可以通过两种方法实现，一种是在路由表设计时直接添加，另一种是在控制器类中添加。这里以在控制器类中添加为例，具体代码如下：

文件 `laravel\app\Http\Controllers\HomeController.php`

```
<?php namespace App\Http\Controllers;
class HomeController extends Controller
{
    public function __construct()    {
        $this->middleware('auth',['only' => ['index']]);
    }
    public function index()
    {
        return view('home');
    }
}
```

这里以 Laravel 5.0 自带的一个控制器类为例，在控制器类中通过 `middleware()` 方法添加了中间件“auth”，该方法可以只有一个参数，即中间件的名称，也可以有两个参数，其中第二个参数用于设定此中间件起作用的控制器方法，该参数为一个关联数组，键值可以为 `only` 或 `except`，`only` 表示设置中间件起作用的方法，`except` 用于设置中间件不起作用的方法。通过上面设置中间件，用户请求首先需要经过中间件来处理，之后由该控制器的 `index()` 方法处理，这里的 `auth` 中间件指的是 `app\Http\Middleware\Authenticate` 类，这种关系是在 `app\Http\Kernel` 类的 `$routeMiddleware` 属性中设定的。接下来介绍中间件对请求进行的处理，具体代码如下：

文件 `laravel\app\Http\Middleware\Authenticate.php`

```
<?php namespace App\Http\Middleware;
use Closure;
use Illuminate\Contracts\Auth\Guard;
class Authenticate
{
    // 创建一个新的过滤实例
    public function __construct(Guard $auth)
    {
        $this->auth = $auth;
    }
    // 处理一个输入请求
    public function handle($request, Closure $next)
    {
        if ($this->auth->guest()) {
            if ($request->ajax()) {
```



```

        return response('Unauthorized.', 401);
    } else {
        return redirect()->guest('auth/login');
    }
}

return $next($request);
}
}

```

在中间件的构造函数中，通过依赖注入的方法完成 \$auth 属性的赋值。这里解决的依赖类型为 Illuminate\Contracts\Auth\Guard，这是一个接口，在 IOC 容器中这种类型的依赖被定义了别名 “auth.driver”，别名定义的过程是在 Illuminate\Foundation\Application 类的注册过程中实现的，这部分在请求到响应的生命周期中也介绍过，于是最终需要注入的依赖为 “auth.driver”，而该依赖是在服务提供者 Illuminate\Auth\AuthServiceProvider 类中注册的，具体代码如下：

文件 Illuminate\Auth\AuthServiceProvider.php

// 注册认证服务

```

protected function registerAuthenticator()
{
    $this->app->singleton('auth', function ($app) {
        $app['auth.loaded'] = true;
        return new AuthManager($app);
    });
    $this->app->singleton('auth.driver', function ($app) {
        return $app['auth']->driver();
    });
}

```

在服务提供者中可以看到，依赖 “auth.driver” 是通过 “\$app['auth']->driver()” 解决的，而 “\$app['auth']” 对应的是 Illuminate\Auth\AuthManager 类实例，也在该注册方法中进行了注册。接下来进一步查看 driver() 方法返回的内容，具体代码如下：

文件 Illuminate\Support\Manager.php

// 获取驱动实例，该类为 Illuminate\Auth\AuthManager 的父类

```

public function driver($driver = null)
{
    $driver = $driver ?: $this->getDefaultDriver();
    if ( ! isset($this->drivers[$driver])){
        $this->drivers[$driver] = $this->createDriver($driver);
    }
    return $this->drivers[$driver];
}

```

文件 Illuminate\Auth\AuthManager.php

// 创建一个新的驱动实例

```
protected function createDriver($driver)
{
    $guard = parent::createDriver($driver);
    $guard->setCookieJar($this->app['Cookie']);
    $guard->setDispatcher($this->app['events']);
    return $guard->setRequest($this->app->refresh('request', $guard,
'setRequest'));
```

文件 Illuminate\Support\Manager.php

```
protected function createDriver($driver)
{
    $method = 'create'.ucfirst($driver).'Driver';
    if (isset($this->customCreators[$driver])){
        return $this->callCustomCreator($driver);
    }
    elseif (method_exists($this, $method)){
        return $this->$method();
    }
    throw new InvalidArgumentException("Driver [$driver] not
supported.");
}
```

文件 Illuminate\Auth\AuthManager.php

// 创建一个 Eloquent 驱动的实例

```
public function createEloquentDriver()
{
    $provider = $this->createEloquentProvider();
    return new Guard($provider, $this->app['session.store']);
}
```

通过以上步骤实际上是创建一个 Illuminate\Auth\Guard 类的实例，该类需要实现 Illuminate\Contracts\Auth\Guard 接口，因为在依赖注入时已经对实例对象的类型进行了限制，即实现上述接口的是类实例对象，换一种说法就是上述接口定义了用户权限认证的所需方法，任何实现该接口的类实例都可以作为依赖注入到路由中间件中进行用户权限认证，在默认情况下先使用 Eloquent ORM 作为数据库驱动，如果需要改为使用其他的驱动，哪怕是以文件记录的用户，只要实现了上述接口，就可以作为用户权限认证模块以供使用，这也是 Laravel 框架松耦合、易扩展的原因，即面向接口编程。下面介绍对用户进行权限认证的方法，具体代码如下：

文件 Illuminate\Auth\Guard.php

```
// 判断用户是否为客人，即未登录认证
public function guest()
{
    return ! $this->check();
}

// 判断当前用户是否已经认证
public function check()
{
    return ! is_null($this->user());
}

// 获取当前认证的用户
public function user()
{
    if ($this->loggedOut) return;
    if ( ! is_null($this->user)){
        return $this->user;
    }

    $id = $this->session->get($this->getName());
    $user = null;
    if ( ! is_null($id)){
        $user = $this->provider->retrieveById($id);
    }

    $recaller = $this->getRecaller();
    if (is_null($user) && ! is_null($recaller)){
        $user = $this->getUserByRecaller($recaller);
        if ($user){
            $this->updateSession($user->getAuthIdentifier());
            $this->fireLoginEvent($user, true);
        }
    }

    return $this->user = $user;
}
```

在路由中间件中是通过 Illuminate\Auth\Guard 类实例的 guest() 方法判断用户是否经过权限认证，而该方法最终是通过 user() 方法获取当前的用户，如果为空则表示用户没有通过权限认证，即未登录；如果能够查找到该用户，则表明该用户已经登录了。对于当前用户的查找是通过 session 机制完成的，关于 Laravel 的 session 机制可以参看第 12 章。

2. 用户权限认证的其他方法

前面介绍了在控制器类中添加路由中间件的方法来实现用户权限认证的实现细节，当然还有其他方法可以实现相同的功能，这里简要介绍其他方法，因为它们的底层实现都是通过 Illuminate\Auth\Guard 类实例来实现的，最终也是基于 session 机制完成的。

(1) 通过路由保护的方式。

```
Route::get(home, ['middleware' => 'auth', 'uses' => 'HomeController@index'] );
```

这种方式与前面通过路由中间件的实现形式一样，在控制器类中添加路由保护中间件可以同时保护很多方法，也可以保护整个控制器类中的所有方法不被未通过权限认证的用户访问，而这种方式只能保护一条路由，当然也可以通过路由组的方式同时保护多条路由，具体代码如下：

```
Route::group(['middleware' => 'auth'], function()
{
    Route::get('home', 'HomeController@index');
    Route::get('home/user', 'HomeController@user');
});
```

(2) 通过获取用户认证信息的方式。

```
public function user()
{
    // Auth::user() 返回认证用户的实例，该实例为 App\User 模型类的实例
    if (Auth::user()){
        // 通过权限认证的用户可以访问
    }
}
```

这种方式是通过 Facades 直接获取用户信息的方法，如果能够获取到用户实例对象，则说明用户已经通过权限认证，否则没有。这种直接获取用户信息的方法也可以通过其他方式，如请求的 Facades 方式和依赖注入的方式。

```
// 通过请求的 Facades 方式
public function user()
{
    // $request->user() 返回认证用户的实例，该实例为 App\User 模型类的实例
    if (Request::user()){
        // 通过权限认证的用户可以访问
    }
}

// 通过依赖注入的方式
<?php namespace App\Http\Controllers;
use Illuminate\Routing\Controller;
use Illuminate\Contracts\Auth\Authenticatable;
class HomeController extends Controller
{
    public function user(Authenticatable $user)
```

```

{
    if($user){
// 通过权限认证的用户可以访问
    }
}
}

```

(3) 直接判断用户是否已认证的方式。

这种方式的实现代码如下：

```

if (Auth::check()){
// 通过权限认证的用户可以访问
}

```

14.2 数据验证

数据验证作为 Web 应用中必不可少的部分，通常用来对数据的正确性进行分析。Laravel 框架中提供了 Validation 类用于验证数据的正确性，如果验证数据错误，则给出相应的错误信息。

14.2.1 数据验证的实现

数据验证一个简单的应用是在用户注册过程中，需要对用户输入的表单信息进行验证，以上一节的用户注册为例，需要验证 E-mail 符合邮件的格式、在 users 表中的唯一性和各表单输入的字符数等。下面以 Laravel 框架中用户认证类（AuthController 类）中的数据验证为例，介绍数据验证实现的细节，具体代码如下：

文件 laravel\app\Http\Controllers\Auth\AuthController.php

```

// 根据注册请求输入，获取验证实例
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
}

```

文件 Illuminate\Foundation\Auth\RegistersUsers.php

```

// 针对应用处理一个注册请求
public function postRegister(Request $request)

```

```

{
    $validator = $this->validator($request->all());
    if ($validator->fails()) {
        $this->throwValidationException(
            $request, $validator
        );
    }
    Auth::login($this->create($request->all()));
    return redirect($this->redirectPath());
}

```

在用户注册的控制器方法中，通过“`$validator = $this->validator($request->all());`”将请求的参数作为验证数据传递给验证方法 `validator()`，该函数返回一个验证实例，可以通过“`$validator->fails()`”来判断数据验证是否正确。下面进一步分析数据是如何进行验证的，具体代码如下：

文件 `Illuminate\Validation\ValidationServiceProvider.php`

```

// 注册一个认证工厂
protected function registerValidationFactory()
{
    $this->app->singleton('validator', function($app)
    {
        $validator = new Factory($app['translator'], $app);
        if (isset($app['validation.presence'])) {
            $validator->setPresenceVerifier($app['validation.
presence']);
        }
        return $validator;
    });
}

```

文件 `Illuminate\Translation\TranslationServiceProvider.php`

```

public function register()
{
    $this->registerLoader();
    $this->app->singleton('translator', function($app)
    {
        $loader = $app['translation.loader'];
        $locale = $app['config']['app.locale'];
        $trans = new Translator($loader, $locale);
        $trans->setFallback($app['config']['app.fallback_
locale']);
        return $trans;
    });
}

```


文件 Illuminate\Translation\TranslationServiceProvider.php

// 注册翻译行加载

```
protected function registerLoader()
{
    $this->app->singleton('translation.loader', function($app){
        return new FileLoader($app['files'], $app['path.lang']);
    });
}
```

前文是通过外观的方式实现输入数据的验证，即代码“Validator::make()”部分，而外观类 Validator 提供的服务名称为“validator”，该服务是通过服务提供者类 ValidationServiceProvider 注册的，用于获取一个验证工厂实例，然后调用验证工厂实例的 make() 方法创建一个验证类实例，具体代码如下：

文件 Illuminate\Validation\Factory.php

// 创建一个新的验证类实例

```
public function make(array $data, array $rules, array $messages = array(),
array $customAttributes = array())
{
    $validator = $this->resolve($data, $rules, $messages,
$customAttributes);
    if ( ! is_null($this->verifier)){
        $validator->setPresenceVerifier($this->verifier);
    }
    if ( ! is_null($this->container)){
        $validator->setContainer($this->container);
    }
    $this->addExtensions($validator);
    return $validator;
}

protected function resolve(array $data, array $rules, array $messages,
array $customAttributes)
{
    if (is_null($this->resolver)){
        return new Validator($this->translator, $data, $rules,
$messages, $customAttributes);
    }
    return call_user_func($this->resolver, $this->translator, $data,
$rules, $messages, $customAttributes);
}
```

通过 make() 方法创建了一个验证类实例，该实例中包含了要验证的数据、准则和验证

过程中需要的其他依赖，并且实例中包含了关于验证的很多方法。下面介绍数据是否符合验证准则的判断方法。

文件 Illuminate\Validation\Validator.php

```
// 判断数据是否符合验证准则
public function fails()
{
    return ! $this->passes();
}

// 判断数据是否通过了验证准则
public function passes()
{
    $this->messages = new MessageBag;
    foreach ($this->rules as $attribute => $rules){
        foreach ($rules as $rule){
            $this->validate($attribute, $rule);
        }
    }
    foreach ($this->after as $after){
        call_user_func($after);
    }
    return count($this->messages->all()) === 0;
}

// 针对一个准则验证一个给定的属性
protected function validate($attribute, $rule)
{
    list($rule, $parameters) = $this->parseRule($rule);
    if ($rule == '') {
        return;
    }
    $value = $this->getValue($attribute);
    $validatable = $this->isValidatable($rule, $attribute, $value);
    $normalizedRule = $this->normalizeRule($rule);
    $method = "validate{$normalizedRule}";
    if ($validatable && ! $this->$method($attribute, $value, $parameters,
$this)) {
        $this->addFailure($attribute, $rule, $parameters);
    }
}
```

数据的验证是一条一条进行的，主要需要 Validator 类实例中的两个属性，分别是 \$data 和 \$rules 两个数组，其中 \$data 中存储的是用户的输入数据，而 \$rules 中存储的是对应数据的准则，如果以前面介绍的注册表单中的 E-mail 数据验证为例，在数据数组中包含

“email” => “1234@163.com”，在数据准则数组中包含 “email” => array(“required”, “email”, “max:255”, “unique:users”)”，然后分别对 “required”、“email”、“max:255” 和 “unique:users” 四个准则进行判断，其中 “unique:users” 准则的验证需要对数据库进行操作。

14.2.2 数据验证的其他使用方法

前面介绍了数据验证的基本方法，还有其他的方法可以简化数据验证的代码量。下面介绍几种更加简捷的验证方法，为软件开发节省时间。这部分内容在官方文档中也有介绍，这里只是进行了归类解释。

1. 控制器验证

大部分的数据验证都是在控制器类中完成的，如果每次都需要手动添加数据、判断数据验证结果并做出相应的动作将需要编写大量的重复代码，会使工作变得很麻烦，而程序员的目标之一就是消除重复。好在 Laravel 框架已经帮我们把重复的部分解决了，在 App\Http\Controllers\Controller 基类使用了一个 ValidatesRequests 的 trait，其中包含的 validate() 函数用于完成数据验证结果的判断、错误信息存储及重定向。针对上节中的实例，可以使用如下方式实现：

```
public function postRegister(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
    Auth::login($this->create($request->all()));
    return redirect($this->redirectPath());
}
```

通过上述代码，如果数据验证通过了，则会继续执行后面的注册用户信息的数据库添加及重定向；如果验证失败则会抛出一个异常并重定向到用户上一次访问的页面，错误信息也会存储至 session 中。如果是 Ajax 请求则不会重定向，而是返回一个状态码为 422 的响应。

2. 表单请求验证

在某些情况下，一个数据验证准则可能在很多地方都需要用到，这时可以使用表单请求验证的方法以消除数据验证准则的重复书写，还有一个目的，就是消除重复、简化代码。可以使用 artisan 命令 “php artisan make:request RegisterRequest” 在 app/Http/Requests 目录下创建一个表单请求验证类，该类中存在一个 rules() 方法，在此方法中可以定义验证规则，针对用户注册的实例可以定义规则如下：

文件 `laravel\app\Http\Requests\RegisterRequest.php`

// 为请求应用提供验证准则

```
public function rules()
```

```
{
```

```
    return [
```

```
        'name' => 'required|max:255',
```

```
        'email' => 'required|email|max:255|unique:users',
```

```
        'password' => 'required|confirmed|min:6',
```

```
    ];
```

```
}
```

上述表单请求验证类提供的验证准则，可以应用在所有控制器方法中作为输入参数约束，通过依赖注入的参数需要通过 `rules()` 方法中的验证准则才能继续执行控制器中的方法，如果验证失败，同样会重定向到上一个页面，错误信息存储在 `session` 中。实现代码如下：

文件 `Illuminate\Foundation\Auth\RegistersUsers.php`

```
public function postRegister (RegisterRequest $request)
```

```
{
```

```
    // 输入的请求已经通过验证了
```

```
}
```

在表单请求类中还包含了另一个 `authorize()` 方法。通过该方法可以实现对用户权限的精确认证控制，即用户是否有权限操作数据库中某个表或字段等。具体代码如下：

文件 `laravel\app\Http\Requests\RegisterRequest.php`

// 判断用户是否有权限实现该请求

```
public function authorize()
```

```
{
```

```
    $article_id = $this->route('article');
```

```
    return Articles::where('id', $article_id)
```

```
        ->where('user_id', Auth::id())->exists();
```

```
}
```

当用户访问由该表单验证请求类约束的控制器方法时，如果对访问的内容没有权限，则无法进行下一步操作。这里首先通过 `route()` 方法来获取请求中 URI 的参数，然后通过数据库查询获取相应 ID 号的文章，最后判断该用户是否具有操作该文章的权限，如果有权限则返回 `true`，否则返回 `false`，最后一个状态为 403 的 HTTP 响应会被返回。

14.2.3 数据验证后期处理

数据验证首先是获取数据验证类实例（`Validator` 类实例），该实例中包含了数据验证需要的所有内容，然后通过 `fails()` 或 `passes()` 方法对数据进行验证。在很多情况下，不仅需要

对数据进行验证，还需要将验证的结果输出给用户，用以提醒用户进行相应的修改。这时就需要将错误信息进行存储并输出，好在 Laravel 框架已经做好了这些，如果想要了解实现细节，可以查看相关源码。实例代码如下：

文件 Illuminate\Foundation\Auth\RegistersUsers.php

```
public function postRegister(Request $request)
{
    $validator = $this->validator($request->all());
    if ($validator->fails()) {
        return redirect('register')->withErrors($validator);
    }
    Auth::login($this->create($request->all()));
    return redirect($this->redirectPath());
}
```

一般情况下，可以通过 “\$errors = \$validator->messages();” 获取数据验证的错误内容，可以简单地将其作为参数输出到视图，Laravel 还提供了一种更简单的方法，如上面实例所示，通过 withErrors() 方法，该方法会将错误信息存储到 session 中，当重定向到其他请求时，会自动将 session 数据中的错误信息读取并以 \$errors 变量加载到视图中，也就是说，可以直接使用该变量设计视图，如 “echo \$messages->first('email');”。

14.2.4 数据验证准则

在 Laravel 框架中默认定义了很多数据验证准则，下面列举出一些常见的准则和实例，当一个数据需要多个准则进行验证时，准则之间以 “|” 分隔。具体准则如表 14.1 所示。

表 14.1 数据验证的具体准则

准则格式	准则内容	实例
email	字段值需符合 E-mail 格式	"email"=>"email"
exists: table,column	字段值需与存在于数据库 table 中的 column 字段值其一相同	"user_name"=>"exists:users,name"
digits: value	字段值需为数字且长度需为 value	"phone_num"=>"digits:11"
digits_between: min,max	字段值需为数字且长度需介于 min 与 max 之间	"tel"=>"digits_between:6,7"
between: min,max	字段值需介于指定的 min 和 max 之间。字符串、数值或文件都是用同样的方式来进行验证	"age"=>"between:0,150"

续表

准则格式	准则内容	实 例
confirmed	字段值需与对应的字段值相同。例如，验证 password 字段，对应的字段 password_confirmation 必须存在且与 password 字段相符	"password"=>"confirmed"
image	文件必须为图片 (JPEG、PNG、BMP、GIF 或 SVG)	"image"=>"image"
in: foo,bar,...	字段值需符合事先给予的清单中的其中一个值	"city"=>"in:Beijing,shanghai"
integer	字段值需为一个整数值	"age"=>"integer"
ip	字段值需符合 IP 格式	"ip"=>"ip"
max: value	字段值需小于等于 value。字符串、数字和文件用于判断 size 大小	"age"=>"max:150"
mimes: foo,bar,...	文件的 MIME 类需在给定的清单的列表中才能通过验证	"photo"=>"mimes:jpeg,bmp,png"
min: value	字段值需大于等于 value。字符串、数字和文件用于判断 size 大小	"age"=>"min:0"
numeric	字段值需为数字	"age"=>"numeric"
regex: pattern	字段值需符合给定的正规表达式	"username"=>"regex:/^/ "
required	字段值为必填	"username"=>"required"

用户权限认证和数据验证是服务端程序必不可少的部分，Laravel 框架提供了相当完备的实现方案供开发者使用，对于一般的小型项目基本已经够用了，但对于复杂的项目则需要进一步开发扩展。Laravel 框架不仅提供了一些组件可以应用，还提供了程序编写的方式和模块功能实现的方式，这些才是它的精髓，在此基础上开发扩展，会以一种全新的方式组织程序代码，从而提高构建大型程序的能力。

第 15 章

思维笔记实例

前面对 Laravel 框架的大多数组件进行了详细的介绍，从它的使用到具体代码的实现细节都有所涉及，但知识和实践之间总是隔着一条鸿沟，阻挡了大部分人对知识升华的脚步。本章将设计一个简单的实例，将之前学习过的不同模块综合起来使用，进而实现某一项基本的应用功能。本章的实例以思维笔记为例，是一个简单的笔记记录软件，可以在日常生活中记录一些学习笔记，功能比较简单，如果需要可以继续拓展其他功能。

虽然目的已经明确，是设计一个笔记管理的软件，那么该从何处入手、分为几个步骤实现还是一个需要思考的问题。开发一个项目可能有不同的流程，如瀑布流、敏捷开发和测试开发等。瀑布流笼统地讲就是按需求分析、方案设计、程序开发、软件测试和软件部署每个环节顺序执行，每个环节结束后进行评审，如果不通过则需要返回重新整改。目前瀑布流开发已经不适应现实中快速开发、快速部署的要求了，更不适合我们这种小的学习项目的设计。我们的设计思路是从简单、直观的环节入手设计程序，一般情况下分为两种，一种是对业务已经基本掌握，改动不会太大，则可以从数据库设计入手；另一种是对需求和业务的了解还不够清楚，需要在实践中进一步理解，则可以从用户界面入手。对于本章中这个项目，由于项目较小，功能单一，可以从数据库设计入手，然后是路由设计，最后是控制器设计和 Web 页面设计。下面将对这些环节逐个进行介绍。

15.1 数据库设计

数据库的设计在项目中可以分为两个部分，一个是数据库中数据表的设计，另一个是对数据库中各数据表操作的接口设计。这两部分内容 Laravel 框架都提供了完美的基础支持，下面分别介绍这两部分内容。

15.1.1 数据表设计

在进行数据库设计时，一般是通过数据库设计软件（如 Power Designer 等）完成数据库的结构设计，之后再通过 SQL 语句完成数据表的创建，但是这种方法在数据表创建和后期

修改时都比较麻烦，效率低下。Laravel 框架为我们提供了数据迁移工具，可以实时管理和更新数据库中各数据表的结构，为数据库的版本控制和管理提供了极大的方便。

首先，根据应用确定需要哪些数据表及每个数据表对应业务所需要的结构。简单的笔记系统只需要三个表就可以满足需求，第一个是用户表，主要用于记录用户信息，可以单人或多人共同使用；第二个是分类表，用于记录笔记中不同的类别，比如学习笔记中关于学科、技术、项目等分类；第三个是笔记内容记录表，用于记录笔记中的主要内容。根据以上分析，设计相应数据表的字段结构，并在 Laravel 框架中设计数据库迁移文件，可以通过 artisan 命令“php artisan make:migration create_users_table”直接生成，并在其中进行修改。数据库迁移文件的具体代码如下：

文件 mindlevel\database\migrations\2015_10_12_000000_create_users_table.php

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateUsersTable extends Migration {
    // 执行数据库迁移
    public function up()
    {
        // 建立用户表
        Schema::create('users',function(Blueprint $table){
            $table->increments('id');
            $table->string('username',32);
            $table->string('account',32);
            $table->string('password',60);
            $table->rememberToken();
            $table->unsignedInteger('addtime');
            $table->tinyInteger('state')->unsigned()->default(1);
        });
        // 建立分类表
        Schema::create('tb_category',function(Blueprint $table){
            $table->increments('id');
            $table->string('name',32);
            $table->unsignedInteger('count')->default(0);
            $table->integer('uid'); // 用户的 id
            $table->tinyInteger('state')->unsigned()-
>default(1); // 分类的状态，确认是否被删除，1 为正常，0 为删除
        });
        // 建立笔记内容表
        Schema::create('tb_records',function(Blueprint $table){
            $table->increments('id');
            $table->string('title',64);
            $table->text('content')->nullable();
```

```
$stable->integer('cid'); // 笔记类型的 id
$stable->integer('uid'); // 用户的 id
$stable->unsignedInteger('addtime')->default(0);
$stable->tinyInteger('state')->default(1)-
>unsigned();

});

}

// 返回数据库迁移
public function down()
{
    Schema::drop('tb_category');
    Schema::drop('tb_records');
    Schema::drop('users');
}

}
```

设计好数据迁移文件后可以通过 artisan 命令“php artisan migrate”完成数据库迁移，迁移后数据库 mindlevel 中的数据表内容如图 15.1 所示。

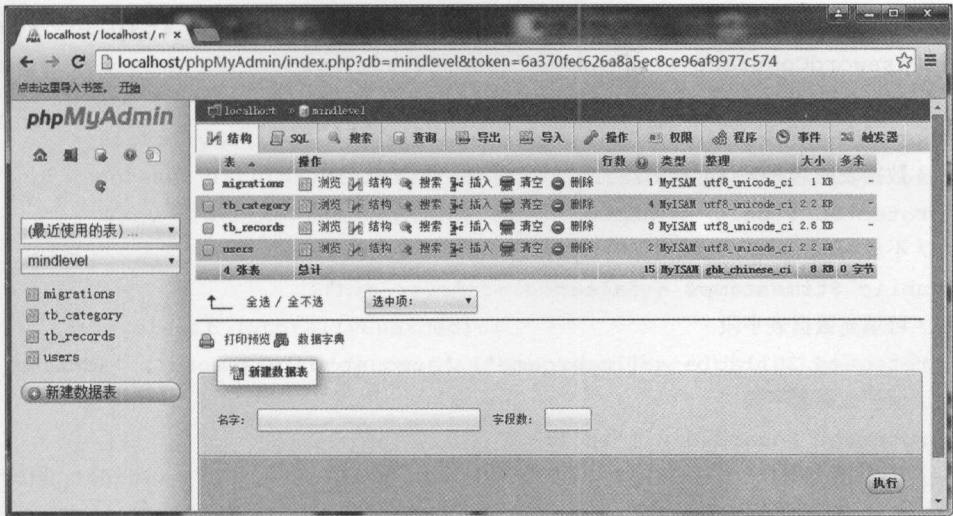


图 15.1 数据库 mindlevel 中的数据表内容

通过上述数据库迁移文件创建三个数据表，具体的表结构为：users 表中有 7 个字段，分别是 id 字段、用户名字段、账户字段、密码字段、登录记录 Token 字段、时间字段和状态字段，这里的时间字段使用无符号整数进行记录；tb_category 表有 5 个字段，分别是 id 字段、类别名字段、笔记数量字段、用户 id 字段和状态字段，其中状态字段为 1 表示正常，为 0 表示该分类已经被删除，在实际项目中，一般的删除行为是不推荐将数据从数据表中删除的，而是在状态字段通过设定状态来标识是否被删除，这样被误删除的内容具有可恢复性；tb_records 表中有 7 个字段，分别是 id 字段、标题字段、内容字段、笔记类型字段、用户 id 字段、创建时间字段和状态字段。

15.1.2 模型类设计

数据库操作的接口可以通过 Laravel 框架提供的 Eloquent ORM 模块进行设计，其中在数据库设计中设计了三个数据表，于是需要为每个数据表建立一个模型类文件与其对应，通常情况下模型类直接放在 `mindlevel\app` 文件夹中，为了程序代码更加模块化，在该文件夹中创建一个 `Models` 文件夹，用来存放相应的模型类。创建的模型类虽然已经具备了对数据库中对应数据表的增加、删除、修改和查询的所有功能，但是在项目开发时还需要一些常用的数据处理接口，可以在模型类中直接添加。具体三个模型类的代码如下：

文件 `mindlevel\app\Models\Users.php`

```
<?php      namespace App\Models;
use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class Users extends Model implements AuthenticatableContract,
CanResetPasswordContract
{
    use Authenticatable, CanResetPassword;
    // 数据表的名称
    protected $table = 'users';
    // 不使用 Laravel 框架提供的时间存储方式
    public $timestamps = false;
    // 可填充数据表字段
    protected $fillable = ['username', 'account', 'password', 'addtime'];
    // 保护数据表字段
    protected $guarded = ['id'];
    // 查找所有用户
    public function findAll()
    {
        $userList = $this->all();
        return $userList;
    }
}
```

这里定义了 `users` 数据表的模型类，其中重新定义了几个字段，`$fillable` 和 `$guarded` 两个字段不是很容易从字面上理解其本质，这里从源码实现的角度来介绍。在项目实现中，如果需要向数据表中插入一行数据，那么方便的做法是通过模型类中的 `fill(array $attributes)` 方法（如果使用 `create()` 方法也会调用 `fill()` 方法），该方法接收的参数是一个数组，可以将数据表相应字段的数据建立一个关联数组传给该方法，在该方法中并不是传入任何数据

都会在表中进行存储，而是首先对数据的合理性进行过滤，满足要求的添加到模型类实例的 `$attributes` 属性数组中，这里的条件就是通过 `$fillable` 和 `$guarded` 两个属性设置的，其中 `$guarded` 定义保护对象，默认值是 “*”，即保护所有，而 `$fillable` 标识可以填充对象，源码中的逻辑是对传入参数数组中每个关联键值通过 `isFillable()` 函数进行判断，即在 `$fillable` 属性数组中查找，如果具有相应的值，则进行添加，否则将舍弃，当舍弃时会判断是否所有数据都保护，如果所有数据都保护就会抛出异常。这里判断所有数据是否都保护是通过 `totallyGuarded()` 函数实现的，该函数判断如果 `$fillable` 属性中没有值并且 `$guarded` 属性中值为 “*” 则全局保护，两者是 “与” 的逻辑，也就是说只要 `$fillable` 中有值或者 `$guarded` 中值不为 “*” 就不会因为全体保护而抛出异常。一般情况下，两个值都使用默认值，即全体保护，在程序设计中只能通过逐个添加字段数据的方式进行数据表中数据的添加，或者将 `$fillable` 和 `$guarded` 两个属性设置成与数据表相符的形式，其中 `$fillable` 中添加需要向数据表中手动添加的字段，而 `$guarded` 设置成不需要向数据表中自动添加的字段，如 “id” 字段。下面给出 `fill()` 函数的源码来增加理解。

文件 `Illuminate\Database\Eloquent\Model.php`

```
// 填充 model 类实例的 attributes 数组
public function fill(array $attributes)
{
    $totallyGuarded = $this->totallyGuarded();
    foreach ($this->fillableFromArray($attributes) as $key => $value) {
        $key = $this->removeTableFromKey($key);
        if ($this->isFillable($key)) {
            $this->setAttribute($key, $value);
        } elseif ($totallyGuarded) {
            throw new MassAssignmentException($key);
        }
    }
}
return $this;
}
```

文件 `mindlevel/app/Models/TbCategory.php`

```
<?php namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class TbCategory extends Model
{
    protected $table = 'tb_category';
    public $timestamps = false;
    protected $fillable = ['name', 'count', 'uid', 'state'];
    protected $guarded = ['id'];
}
```

// 定义类型的状态

```

static $state = array('use' => 1, 'delete' => 0);

// 加载所有笔记类别信息
public function findUserAll($uid = 0)
{
    if($uid!=0){
        $st = self::$state['use'];
        $catArr = static::whereRaw("uid={$uid} and state={$st}")->get()-
>toArray();
        return $catArr;
    }else{
        return null;
    }
}

// 查找 $uid 的第一个分类或者创建一个, 返回 id 号
public function firstOrNewId($uid)
{
    $st = self::$state['use'];
    $modColl = static::whereRaw("uid={$uid} and state={$st}")->get();//
builder
    if($modColl->count()>0){
        $id = $modColl->first()->getAttribute('id');
        return $id;
    }else{
        return $this->createNewCat($uid);
    }
}

// 添加一个新闻类别
public function categoryAdd($param)
{
    $param['name'] = isset($param['name'])? $param['name']:' 思维笔记 ';
    if(empty($param['uid'])){
        return;
    }
    $this->create([
        'name' => $param['name'],
        'count' => 0,
        'uid' => $param['uid'],
        'state' => self::$state['use'],
    ]);
}

// 删除本用户的一个分类, 返回下一个该用户的分类, 并将日记添加到其中
public function deleteCat($uid, $cid)
{

```



```

$model = $this->find($cid);
$model->state = self::$state['delete'];
$model->save();
return $this->firstOrCreate($uid);
}

// 获取笔记类别的第一个 id
public function firstId($uid)
{
    $st = self::$state['use'];
    $modColl = static::whereRaw("uid={$uid} and state={$st}")-
>get();

    if($modColl->count()>0){
        $id = $modColl->first()->getAttribute('id');
        return $id;
    }else{
        return null;
    }
}

// 创建一个新类别
public function createNewCat($uid, $name = null)
{
    $username = $name?'思维笔记':$name;
    $model = $this->create([
        'name' => $username,
        'count' => 0,
    ]);
    return $model['id'];
}

// 分类的数量减去 $num
public function countReduce($cid, $num)
{
    $model = static::find($cid);
    $model->count = $model->count-$num;
    $model->save();
}
}

```

文件 mindlevel\app\Models\TbRecords.php

```

<?php    namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class TbRecords extends Model
{
    protected $table = 'tb_records';
    public $timestamps = false;
}

```

```

        protected $fillable = ['title','content','cid','uid','addtime','state'];

        protected $guarded=['id'];
        // 定义类型的状态
        static $state = array('use' => 1, 'delete' => 0);
        // 执行增加操作
        public function insert($param)
        {
            if(isset($param['uid']) && $param['uid']>0){
                $tbmodel = new TbCategory();
                $param['cid'] = isset($param['cid'])? $param['cid']: $tbmodel->firstOrCreate($param['uid']);
                $this->fill($param);
                $suc = $this->save();
                if($suc){
                    $cat = TbCategory::find($param['cid']);
                    $cat->count = $cat->count+1;
                    return $cat->save();
                }
            }
            return false;
        }
        // 新闻的浏览
        public function findUserAll($uid,$cid = 0)
        {
            $st = self::$state['use'];
            if($cid>0){
                $recordsList = static::whereRaw("uid={$uid} and cid={$cid} and state={$st}")->get()->toArray();
            }else{
                $recordsList = static::whereRaw("uid={$uid} and state={$st}")->get()->toArray();
            }
            return $recordsList;
        }
        // 将用户 $uid 笔记的类型从 $cid 变成 $nextId
        public function changeCat($uid, $cid, $nextId)
        {
            $st = self::$state['use'];
            return static::whereRaw("uid={$uid} and cid={$cid} and state={$st}")->update(['cid'=>$nextId]);
        }
    }

```

15.2 路由设计

15.2.1 模块划分

在路由设计时需要进行模块化设计，否则随着业务的不断增加将会导致路由设计混乱不堪，继而使相应路由入口的权限认证、中间件处理等功能的添加变得困难，最终导致一些安全问题和路由错误。对于思维笔记这个实例，将其划分为三个路由模块，第一个模块是思维笔记展示模块，即在用户没有登录的情况下依然可以看到相应内容；第二个模块是用户认证模块，这部分主要实现用户的注册和登录；第三个模块是思维笔记的管理模块，这部分内容是用户登录后可以操作的内容，主要包括用户的管理、笔记分类的管理和笔记的管理等内容。

15.2.2 程序设计

Laravel 框架提供了多种不同路由设计方式，有些方式使得路由设计简单灵活，如 `get()`、`post()` 等方法，有些方式适合模块化设计，如 RESTful 资源控制器（`resource()` 方法）、隐式控制器（`controller()` 方法）等。这里结合上面两种路由设计方式，可以非常轻松地设计出路由。具体路由由文件代码如下：

文件 `mindlevel\app\Http\routes.php`

```
<?php
// 用于 Web 浏览
Route::get('/', 'WebController@index');
Route::controllers([
    'Web' => 'WebController',
]);
// 用于 Web 认证控制
Route::controllers([
    'webauth' => 'Auth\WebAuthController',
    'webpassword' => 'Auth\WebPasswordController',
]);
```

15.3 控制器设计与 Web 页面设计

控制器设计和 Web 页面设计之间具有一定的耦合性，大型项目需要通过文档约定进行解耦合，从而实现模块化分工设计。本章中的简单实例可以通过迭代的方式进行快速开发，即设计一个页面，再进行控制器内容的设计。这里主要分为几个模块，分别是用户认证模块、用户管理模块、笔记类别管理模块和笔记管理模块，下面将对这四个模块的设计分别进行介绍。

15.3.1 用户认证模块

用户认证模块主要设计了用户注册和登录两个基本功能，Laravel 框架中对用户认证模块各个功能已经提供了基础支持，这里直接使用相应模块实现即可，这部分可以参考认证和数据验证的相关内容。在页面设计时使用 Laravel 框架提供的 blade 模板，建立一个公共的模板文件 `app.blade.php`，用于提供所有视图公共的部分，其他视图页面都继承该视图。下面给出公共视图代码：

文件 `mindlevel/resources/views/app.blade.php`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>MindLevel</title>
    <link href="{{ asset('/css/app.css') }}" rel="stylesheet">
    <link href="//fonts.googleapis.com/css?family=Roboto:400,300"
rel='stylesheet' type='text/css'>
    <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.
js"></script>
    <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></
script>
</head>
<body>
    <nav class="navbar navbar-default">
        <div class="container-fluid">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1">
                    <span class="sr-only">Toggle Navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="#">MindLevel</a>
            </div>
            <div class="collapse navbar-collapse" id="bs-example-navbar-
collapse-1">
                <ul class="nav navbar-nav">
                    <li><a href="{{ url('/') }}">Web</a></li>
                </ul>
                <ul class="nav navbar-nav navbar-right">
```

```

        @if (Auth::guest())
            <li><a href="{{ url('/webauth/login') }}">Login</a></li>

            <li><a href="{{ url('/webauth/register') }}">Register</a></li>

        @else
            <li class="dropdown">
                <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false">{{ Auth::user()->name }}
                <span class="caret"></span></a>
                <ul class="dropdown-menu" role="menu">
                    <li><a href="{{ url('/webauth/logout') }}">Logout</a></li>
                </ul>
            </li>
        @endif
    </ul>
</div>
</div>
</nav>
@yield('content')
<script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.1/js/bootstrap.min.js"></script>
</body>
</html>

```

下面设计用户登录和注册视图，这些视图在 blade 公共视图布局的基础上设计而成，具体代码如下：

文件 mindlevel/resources/views/auth/login.blade.php

```

@extends('app')
@section('content')
<div class="container-fluid">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">Login</div>
                <div class="panel-body">
                    @if (count($errors) > 0)
                        <div class="alert alert-danger">
                            <strong>Whoops!</strong> There were some problems with
                            your input.<br><br>

```

```

<ul>
    @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>
</div>
@endif
<form class="form-horizontal" role="form" method="POST"
action="{{ url('/webauth/login') }}">
    <input type="hidden" name="_token" value="{{ csrf_token()
}}">
    <div class="form-group">
        <label class="col-md-4 control-label">Account</label>
        <div class="col-md-6">
            <input type="text" class="form-control" name="account"
value="{{ old('email') }}">
        </div>
    </div>
    <div class="form-group">
        <label class="col-md-4 control-label">Password</label>
        <div class="col-md-6">
            <input type="password" class="form-control"
name="password">
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-6 col-md-offset-4">
            <div class="checkbox">
                <label>
                    <input type="checkbox" name="remember">
Remember Me
                </label>
            </div>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-6 col-md-offset-4">
            <button type="submit" class="btn btn-primary">Login</
button>
            <a class="btn btn-link" href="{{ url('/password/email')
}}">Forgot Your Password?</a>
        </div>
    </div>
</form>

```



```

        </div>
    </div>
</div>
</div>
</div>
@endsection
文件 mindlevel\resources\views\auth\register.blade.php
@extends('app')
@section('content')
<div class="container-fluid">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">Register</div>
                <div class="panel-body">
                    @if (count($errors) > 0)
                        <div class="alert alert-danger">
                            <strong>Whoops!</strong> There were some problems with
your input.<br><br>
                            <ul>
                                @foreach ($errors->all() as $error)
                                    <li>{{ $error }}</li>
                                @endforeach
                            </ul>
                        </div>
                    @endif
                    <form class="form-horizontal" role="form" method="POST"
action="{{ url('/webauth/register') }}">
                        <input type="hidden" name="_token" value="{{ csrf_token()
}}">
                        <div class="form-group">
                            <label class="col-md-4 control-label">User Name</label>
                            <div class="col-md-6">
                                <input type="text" class="form-control" name="username"
value="{{ old('username') }}">
                            </div>
                        </div>
                        <div class="form-group">
                            <label class="col-md-4 control-label">Account</label>
                            <div class="col-md-6">
                                <input type="account" class="form-control"
name="account" value="{{ old('account') }}">
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

```

```

        <div class="form-group">
            <label class="col-md-4 control-label">Password</label>
            <div class="col-md-6">
                <input type="password" class="form-control"
name="password">
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-4 control-label">Confirm Password</
label>
            <div class="col-md-6">
                <input type="password" class="form-control"
name="password_confirmation">
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-6 col-md-offset-4">
                <button type="submit" class="btn btn-primary">
                    Register
                </button>
            </div>
        </div>
    </form>
</div>
</div>
</div>
</div>
</div>
@endsection

```

下面针对上述用户认证视图页面设计相应的控制器，用户认证控制器的相关方法在 Laravel 框架中已经通过 `AuthenticatesAndRegistersUsers` 模块设计好了，只需要包含这个 trait 并实现两个相应的接口函数即可，接口函数分别是 `validator()` 函数和 `create()` 函数，一个用于验证注册输入，另一个用于向用户数据表中添加信息，相应代码如下：

文件 mindlevel\app\Http\Controllers\Auth\WebAuthController.php

```
<?php    namespace App\Http\Controllers\Auth;
use App\Models\Users;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;
class WebAuthController extends Controller
{
    use AuthenticatesAndRegistersUsers;
```

```

// 创建一个新的授权控制器实例
public function __construct()
{
    $this->middleware(['guest', ['except' =>
['getLogout','getRegister']]);
}
// 对输入的登录数据进行验证
protected function validator(array $data)
{
    return Validator::make($data, [
        'username' => 'required|max:255',
        'account' => 'required|max:255|unique:users',
        'password' => 'required|confirmed|min:6',
    ]);
}
// 向用户数据表中添加一行数据
protected function create(array $data)
{
    return Users::create([
        'username' => $data['username'],
        'account' => $data['account'],
        'password' => bcrypt($data['password']),
        'addtime' => time(),
    ]);
}
}

```

15.3.2 用户管理模块

用户登录后会进入思维笔记的操作页面，该页面包含用户管理、笔记分类管理和笔记管理的操作，其视图页面如图 15.2 所示。

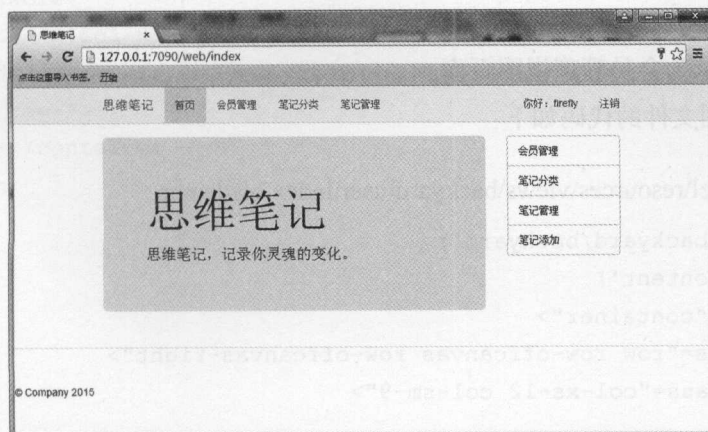


图 15.2 思维笔记操作视图页面

思维笔记操作视图文件的代码如下：

文件 mindlevel\resources\views\backyard\include\sidebar.blade.php

```
<div class="col-xs-6 col-sm-3 sidebar-offcanvas" id="sidebar">
    <div class="list-group">
        <a href="/web/userlist" class="list-group-item ">会员管理 </a>
        <a href="/web/categorylist" class="list-group-item ">笔记分类 </a>
        <a href="/web/noteslist" class="list-group-item ">笔记管理 </a>
        <a href="/web/notesadd" class="list-group-item ">笔记添加 </a>
    </div>
</div><!--/.sidebar-offcanvas-->
```

文件 mindlevel\resources\views\backyard\index\index.blade.php

```
@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
        <div class="col-xs-12 col-sm-9">
            <p class="pull-right visible-xs">
                <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle
                nav</button>
            </p>
            <div class="jumbotron">
                <h1>思维笔记 </h1>
                <p>思维笔记，记录你灵魂的变化。</p>
            </div>
        </div>
        @include('backyard.include.sidebar')
    </div><!--/row-->
</div> <!-- /container -->
@endsection
```

如图 15.3 所示为会员管理视图页面。

会员管理视图文件的代码如下：

文件 mindlevel\resources\views\backyard\user\index.blade.php

```
@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
        <div class="col-xs-12 col-sm-9">
```

```

<p class="pull-right visible-xs">
    <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
</p>
<h3>会员信息列表</h3>
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>ID 号</th>
            <th>账号</th>
            <th>注册时间</th>
            <th>状态</th>
            <th>操作</th>
        </tr>
    </thead>
    <tbody>
        <?php
foreach($attributes['userList'] as $vo){
    echo "<tr>";
    echo "<td>{$vo['id']}</td>";
    echo "<td>{$vo['username']}</td>";
    echo "<td>".date("Y-m-d", $vo['addtime'])."</td>";
    echo "<td>".(($vo['state']==1)?" 启用 ":" 禁用 ")."</td>";
    echo "<td>
        <a href=\""/backyard/userInfo?uid={$vo['id']}\"><span
class=\"glyphicon glyphicon-search\" title=\" 详情 \"></span></a>&nbsp;&nbsp;&nbsp;
        </td>";
    echo "</tr>";
}
?>
</tbody>
</table>
</div><!--/.col-xs-12.col-sm-9-->
@include('backyard.include.sidebar')
</div><!--/row-->
</div> <!-- /container -->
@endsection

```

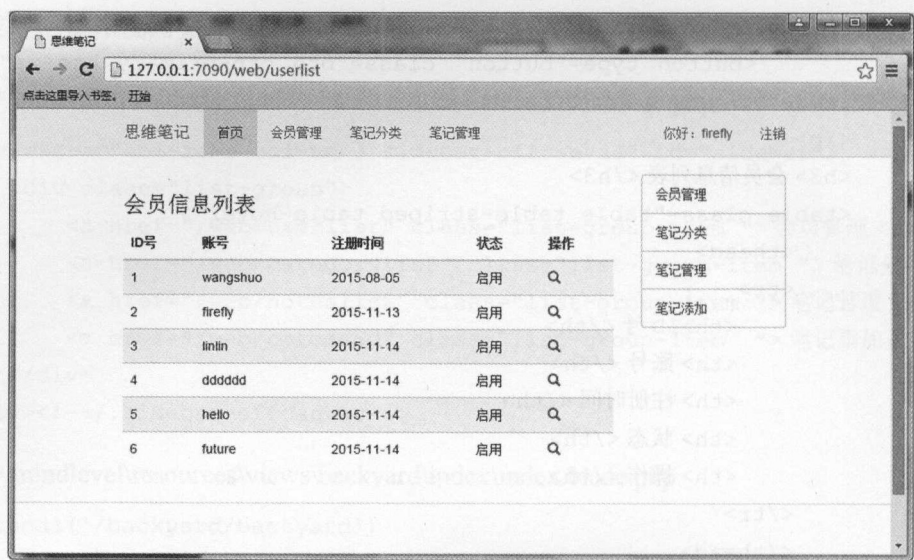


图 15.3 会员管理视图

思维笔记管理的控制器设计通过一个控制器类 (App\Http\Controllers\ WebController 类) 实现，用户管理的控制器方法的设计如下：

```
<?php namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use Illuminate\Contracts\Auth\Guard;
use Request;
use Validator;
use App\Models\Users;
use App\Models\TbCategory;
use App\Models\TbRecords;
class WebController extends Controller
{
    // 自定义模板引擎的属性信息
    private $attributes = array();
    public function __construct(Guard $auth)
    {
        $this->middleware('auth');
        $this->auth = $auth;
        if($this->auth->user()){
            $this->authSave();
        }
    }
    private function authSave()
    {

```



```

        $this->attributes['username'] = $this->auth->user()-
>getAttribute('username');
        $this->attributes['uid'] = $this->auth->id();
    }

    // 查询属性信息
    private function getAttribute($key)
    {
        return $this->attributes[$key];
    }

    // 在 attributes 中放置的属性
    private function addAttributes($param,$value)
    {
        $this->attributes[$param] = $value;
    }

    //Web 访问根目录时的功能
    public function getIndex()
    {
        $user = $this->auth->user();
        return view('/backyard/index/index')->with('attributes',$this-
>attributes);
    }

    // 会员信息列表
    public function getUserlist()
    {
        $module = new Users();
        $data = $module->findAll()->toArray();
        $this->addAttributes("userList", $data);
        return view("/backyard/user/index")->with('attributes',$this-
>attributes);
    }
}

```

15.3.3 笔记类别管理模块

笔记类别管理主要需要实现几个功能，分别是笔记类别显示功能、笔记类别添加功能、笔记类别浏览功能和笔记类别删除功能，其中笔记类别显示功能用于显示该用户的所有笔记类别，以及每个类别中包含多少篇文章，笔记类别的添加和删除功能用于添加和删除新的笔记类别，而笔记类别浏览功能是浏览该类别中所有笔记的标题。笔记类别管理视图页面如图 15.4 所示。

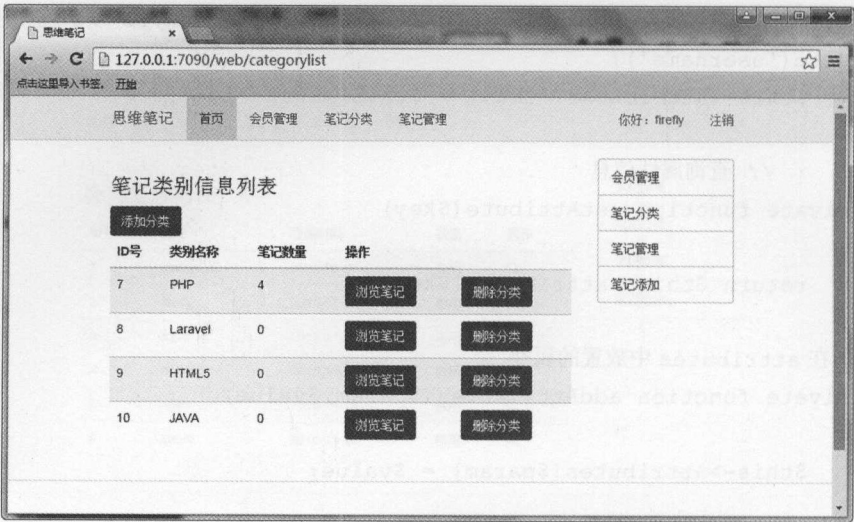


图 15.4 笔记类别管理视图

笔记类别浏览视图页面如图 15.5 所示。

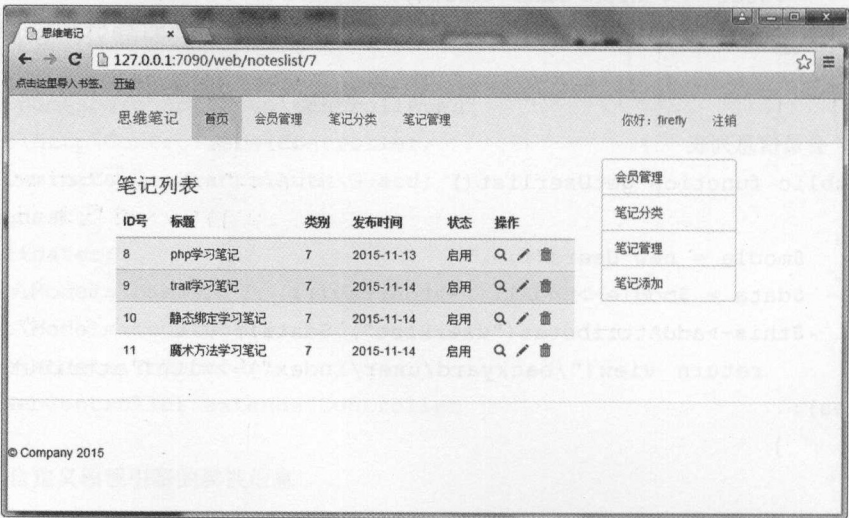


图 15.5 笔记类别浏览视图

笔记类别管理视图文件的代码如下：

```
文件 mindlevel\resources\views\backyard\category\index.blade.php
@extends('/backyard/backyard')
@section('content')
<div class="container">
  <div class="row row-offcanvas row-offcanvas-right">
    <div class="col-xs-12 col-sm-9">
      <p class="pull-right visible-xs">
        <button type="button" class="btn btn-primary btn-xs" data-
```

```

toggle="offcanvas">Toggle nav</button>
</p>
<h3> 笔记类别信息列表 </h3>
    <button type="button" class="btn btn-primary" data-
toggle="offcanvas" onclick="window.location='/web/categoryadd'"> 添 加 分 类 </
button>
    <table class="table table-striped table-hover">
        <thead>
            <tr>
                <th>ID 号 </th>
                <th>类别名称 </th>
                <th>笔记数量 </th>
                <th>操作 </th>
            </tr>
        </thead>
        <tbody>
            <?php
                foreach($attributes['catList'] as $vo){
                    echo "<tr>";
                    echo "<td>{$vo['id']}</td>";
                    echo "<td>{$vo['name']}</td>";
                    echo "<td>{$vo['count']}</td>";
                    echo "<td><button type=\"button\" class=\"btn btn-primary\"
onclick=\"window.location='/web/noteslist/{\$vo['id']}\"> 浏览 笔 记 </button></
td>";
                    echo "<td><button type=\"button\" class=\"btn btn-primary\"
onclick=\"window.location='/web/deletecat/{\$vo['id']}\"> 删 除 分 类 </button></
td>";
                    echo "</tr>";
                }
            ?>
        </tbody>
    </table>
</div><!--/.col-xs-12.col-sm-9-->
    @include('backyard.include.sidebar')
</div><!--/row-->
</div> <!-- /container -->
@endsection

文件 mindlevel\resources\views\backyard\category\add.blade.php

@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">

```



```

<div class="col-xs-12 col-sm-9">
    <p class="pull-right visible-xs">
        <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
    </p>
    <h3> 笔记类别添加 </h3>
    <input type="hidden" id="domain" value="http://7u2r8g.com1.
z0.glb.clouddn.com/">
    <input type="hidden" id="uptoken_url" value="/admin.php/
uptoken">
    <form class="form-horizontal col-sm-10" role="form" action="/
web/categoryadd" method="post">
        <div class="form-group">
            <label for="inputEmail3" class="col-sm-2 control-
label"> 类别名称 </label>
            <div class="col-sm-10">
                <input type="text" name="name" class="form-
control" id="inputEmail3" placeholder="text">
            </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-10">
                <input type="submit" class="btn btn-default"
value=" 添加 " />
            </div>
        </div>
    </form>
</div><!--/.col-xs-12.col-sm-9-->
@include('backyard.include.sidebar')
</div><!--/row-->
</div> <!-- /container -->
@endsection

```

文件 mindlevel/resources/views/backyard/notes/index.blade.php

```

@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
        <div class="col-xs-12 col-sm-9">
            <p class="pull-right visible-xs">
                <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
            </p>
            <h3> 笔记列表 </h3>

```

```
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>ID 号 </th>
            <th>标题 </th>
            <th>类别 </th>
            <th>发布时间 </th>
            <th>状态 </th>
            <th>操作 </th>
        </tr>
    </thead>
    <tbody>
        <?php
        foreach($attributes['recordsList'] as $vo){
            echo "<tr>";
            echo "<td>{$vo['id']}</td>";
            echo "<td>{$vo['title']}</td>";
            echo "<td>{$vo['cid']}</td>";
            echo "<td>".date("Y-m-d",$vo['addtime'])."</td>";
            echo "<td>".(($vo['state']==1)? "启用 ":" 禁用 ")."</td>";

            td>";

            echo "<td>

                <a href=\"/web/noteshow/
{$vo['id']}\"><span class=\"glyphicon glyphicon-search\" title=\" 详 情 \"></
span></a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
                <a href=\"/web/noteedit/
{$vo['id']}\"><span class=\"glyphicon glyphicon-pencil\" title=\" 编 辑 \"></
span></a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
                <a href = \" / web /
deletenote/{$vo['id']}\"><span class=\"glyphicon glyphicon-trash\" title=\" 删
除 \"></span></a>

            </td>";
            echo "</tr>";
        }
        ?>
    </tbody>
</table>
</div><!--/.col-xs-12.col-sm-9-->
@include('backyard.include.sidebar')
</div><!--/row-->
</div> <!-- /container -->
@endsection
```

接下来针对上述需要的功能和视图页面进行控制器方法的设计，这里对于笔记类

别的显示通过 `getCategorylist()` 方法实现，主要是通过用户名 `id` 获取关于该用户的所有笔记分类，并通过 `/backyard/category/index` 视图显示出来；笔记类别的添加是通过 `getCategoryadd()` 方法和 `postCategoryadd()` 方法实现的，其中 `get()` 方法用于显示添加视图页面，而 `post()` 方法用于提交用户添加笔记分类的数据并在相应数据表中进行添加；删除笔记分类是通过 `getDeletecat()` 方法实现的，该方法接收一个分类的 `id` 并将数据表的状态字段设置为 0，表示该分类已经被删除，这里为了不丢失笔记，会将该分类中的笔记添加到其他的分类中。

```
// 显示笔记类别信息
public function getCategorylist()
{
    $model = new TbCategory();
    $data = $model->findUserAll($this->getAttribute('uid'));
    $this->addAttributes("catList", $data);
    return view("/backyard/category/index")->with('attributes', $this->attributes);
}

// 添加笔记类别
public function getCategoryadd()
{
    return view("/backyard/category/add")->with("attributes", $this->attributes);
}

// 添加笔记类别
public function postCategoryadd()
{
    $model = new TbCategory();
    $param = $_POST;
    $param['uid'] = $this->getAttribute('uid');
    $param['count'] = 0;
    $param['state'] = TbCategory::$state['use'];
    $model->categoryAdd($param);
    return redirect("/web/categorylist");
}

// 删除一个分类
public function getDeletecat($cid)
{
    $model = new TbCategory();
    $nextId = $model->deleteCat($this->getAttribute('uid'), $cid);
    $remodel = new TbRecords();
    $num = $remodel->changeCat($this->getAttribute('uid'), $cid, $nextId);
    $nextCat = $model->find($nextId);
```



```
$nextCat->count = $nextCat->count+$num;
$nextCat->save();
return redirect("/web/categorylist");
}
// 加载笔记列表页
public function getNoteslist($cid=0)
{
    $model = new TbRecords();
    $data = $model->findUserAll($this->getAttribute('uid'), $cid);
    $this->addAttributes("recordsList", $data);
    return view("/backyard/notes/index")->with("attributes",$this->attributes);
}
```

15.3.4 笔记管理模块

笔记管理模块主要包含笔记浏览显示、笔记添加、笔记删除、笔记修改和编辑等操作。其中，笔记浏览显示功能将显示所有类别的笔记；在笔记添加功能页面，需要添加笔记的标题和内容；笔记类别是通过选择确定的，如果没有相应的类型需要先添加笔记类别；笔记修改和编辑是将原笔记内容取出并添加到视图中，可以对笔记所有的属性进行修改和编辑。笔记管理视图页面如图 15.6 所示。

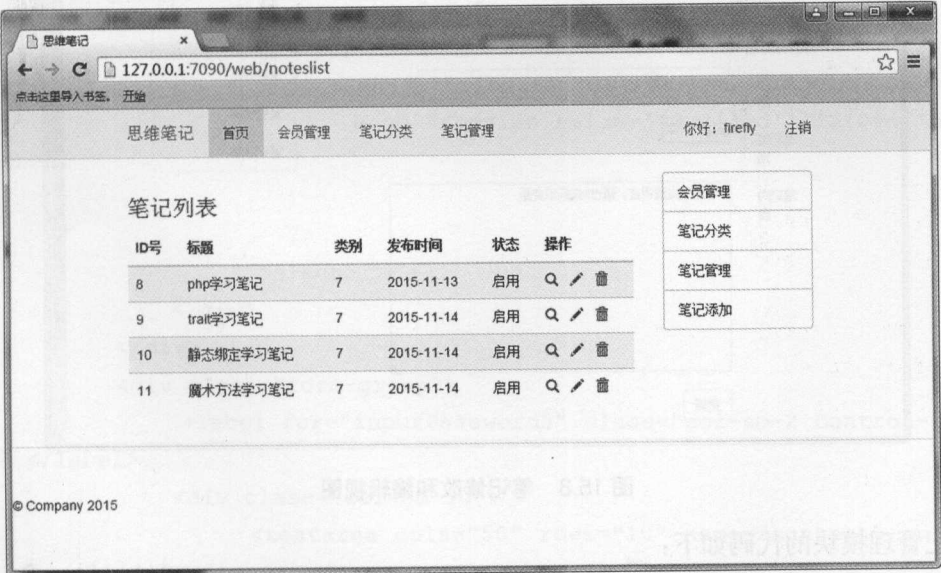


图 15.6 笔记管理视图

笔记添加视图页面如图 15.7 所示。

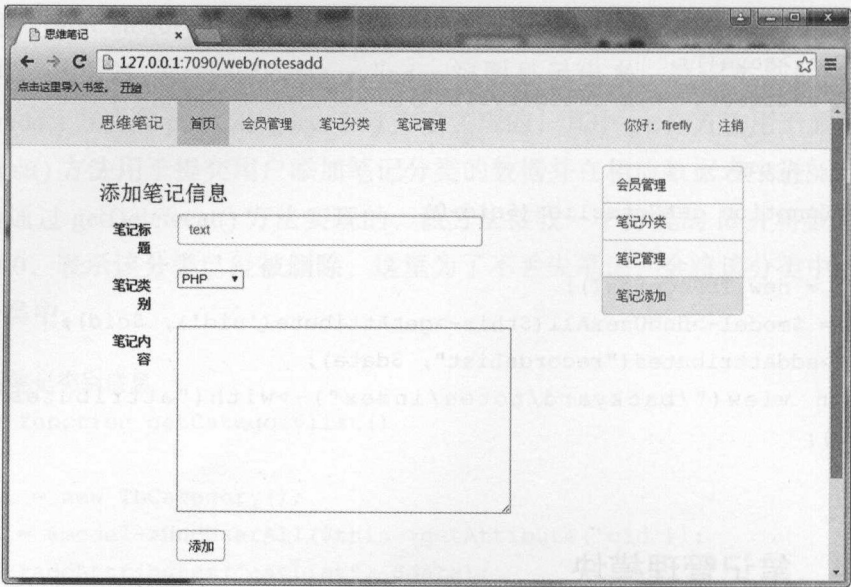


图 15.7 笔记添加视图

笔记修改和编辑视图页面如图 15.8 所示。

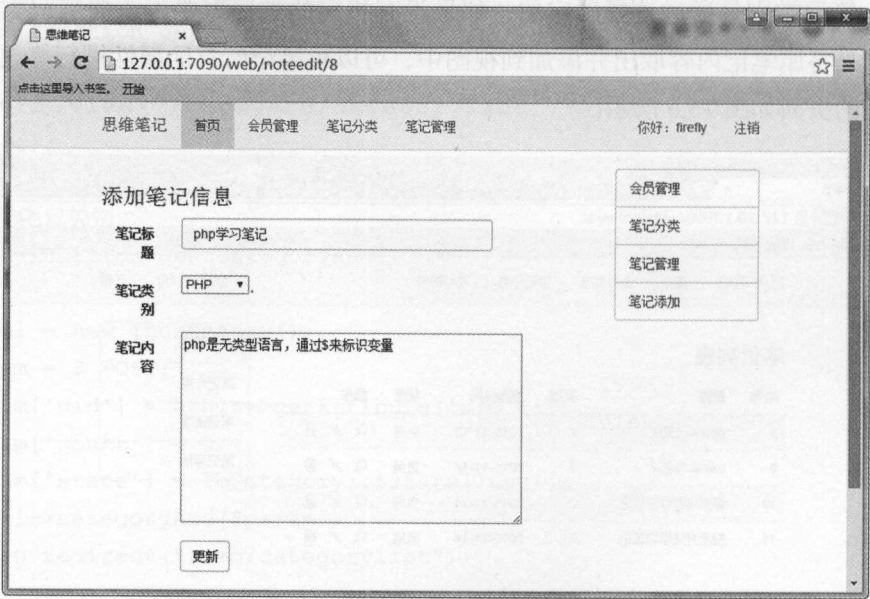


图 15.8 笔记修改和编辑视图

笔记管理模块的代码如下：

文件 mindlevel\resources\views\backyard\notes\add.blade.php

```
@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
```

```

<div class="col-xs-12 col-sm-9">
  <p class="pull-right visible-xs">
    <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
  </p>
  <h3>添加笔记信息</h3>
  <input type="hidden" id="domain" value="http://7u2r8g.com1.z0.glb.
cloudn.com/">
  <input type="hidden" id="uptoken_url" value="/admin.php/uptoken">
  <form class="form-horizontal col-sm-10" role="form" action="/web/
notesinsert" method="post">
    <div class="form-group">
      <label for="inputEmail3" class="col-sm-2 control-label">笔记
标题</label>
      <div class="col-sm-10">
        <input type="text" name="title" class="form-control" id=
"inputEmail3" placeholder="text" value="{{ old('title') }}">
      </div>
    </div>
    <div class="form-group">
      <label for="inputEmail3" class="col-sm-2 control-label">笔记
类别</label>
      <div class="col-sm-10">
        <select name="cid">
          <?php
            foreach($attributes['cglst'] as $vo){
              echo "<option value='{$vo['id']}'>{$vo['name']}"
</option>";
            }
          ?>
        </select>
      </div>
    </div>
    <div class="form-group">
      <label for="inputPassword3" class="col-sm-2 control-label">
笔记内容</label>
      <div class="col-sm-10">
        <textarea cols="50" rows="10" name="content" value="{{
old('content') }}"></textarea>
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-10">
        <input type="submit" class="btn btn-default" value="添加

```



```

/>
        </div>
    </div>
</form>
</div><!--/.col-xs-12.col-sm-9-->
    @include('backyard.include.sidebar')
</div><!--/row-->
</div> <!-- /container -->
@endsection

文件 mindlevel\resources\views\backyard\notes\edit.blade.php

@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
        <div class="col-xs-12 col-sm-9">
            <p class="pull-right visible-xs">
                <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
            </p>
            <h3> 添加笔记信息 </h3>
            <input type="hidden" id="domain" value="http://7u2r8g.com1.z0.glb.
cloudcdn.com/">
            <input type="hidden" id="uptoken_url" value="/admin.php/uptoken">
            <form class="form-horizontal col-sm-10" role="form" action="/web/
notechange/{{${id}}}" method="post">
                <div class="form-group">
                    <label for="inputEmail3" class="col-sm-2 control-label">笔
记标题 </label>

                    <div class="col-sm-10">
                        <input type="text" name="title" class="form-control"
id="inputEmail3" placeholder="text" value="{{ $title }}">
                    </div>
                </div>
                <div class="form-group">
                    <label for="inputEmail3" class="col-sm-2 control-label">笔
记类别 </label>

                    <div class="col-sm-10">
                        <select name="cid" >
                            <?php
                                foreach($attributes['cglist'] as $vo){
                                    if($vo['id'] == $cid){
                                        echo "<option value='{${vo['id']}'
selected>{{${vo['name']}}</option>";

```

```

        }else{
            echo "<option value='{ $vo['id']}'>{ $vo['name']}</option>";
        }
    }
    ?>
</select>
</div>
</div>
<div class="form-group">
    <label for="inputPassword3" class="col-sm-2 control-label">
笔记内容 </label>
    <div class="col-sm-10">
        <textarea cols="50" rows="10" name=
"content">{{ $content }}</textarea>
    </div>
</div>
<div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
        <input type="submit" class="btn btn-default" value="更
新 " />
    </div>
</div>
</form>
</div><!-- /.col-xs-12.col-sm-9-->
@include('backyard.include.sidebar')
</div><!-- /row-->
</div> <!-- /container -->
@endsection

```

文件 mindlevel/resources/views/backyard/notes/show.blade.php

```

@extends('/backyard/backyard')
@section('content')
<div class="container">
    <div class="row row-offcanvas row-offcanvas-right">
        <div class="col-xs-12 col-sm-9">
            <p class="pull-right visible-xs">
                <button type="button" class="btn btn-primary btn-xs" data-
toggle="offcanvas">Toggle nav</button>
            </p>
            <h3>{{ $title }}</h3>
            <p>
                {{ $content }}
            </p>

```

```

        </div><!--/.col-xs-12.col-sm-9-->
        @include('backyard.include.sidebar')
    </div><!--/row-->
</div> <!-- /container -->
@endsection

```

在笔记管理控制器的设计中，笔记列表显示与笔记类别管理中的笔记类别显示使用相同的视图和控制器方法，只是在显示笔记类别时会传递类别号，于是只显示该类别的笔记信息，而在笔记类别管理中的笔记显示则不传递类别号，于是显示所有类别的笔记信息；笔记添加是通过 `getNotesadd()` 方法和 `postNotesinsert()` 方法实现的，同样，`get()` 方法用于显示笔记添加视图页面与用户交互，而 `post()` 方法是将用户提交的数据添加到相应的数据表中，笔记添加需要注意的是验证其添加的内容是否符合数据表设计的要求；笔记的删除是通过 `getDeletenote($id)` 方法实现的，在数据表中并没有真正地将笔记删除，而是将状态标识字段设置为 0；笔记修改和编辑是通过 `getNotecedit($id)` 方法实现的，根据笔记的 id 号取出笔记并进行编辑，然后对数据表中的数据进行更新。笔记管理控制器的设计代码如下：

```

// 添加笔记
public function getNotesadd()
{
    $model = new TbCategory();
    $data = $model->findUserAll($this->getAttribute('uid'));
    $this->addAttributes('cglist', $data);
    return view("/backyard/notes/add")->with("attributes", $this->attributes);
}
// 插入笔记
public function postNotesinsert()
{
    $validator = $this->validator(Request::all());
    if ($validator->fails()) {
        $this->throwValidationException(
            Request::instance(), $validator
        );
    }
    // 准备数据
    $model = new TbCategory();
    $param = $_POST;
    $param['uid'] = $this->getAttribute('uid');
    $param['cid'] = (!empty($param['cid'])) ? $param['cid'] : $model-
>firstOrCreate($this->getAttribute('uid'));
    $param['addtime'] = time();
    $param['state'] = 1;
    // 存储笔记
    $rmodel = new TbRecords();

```



```

    $suc = $rmodel->insert($param);
    if($suc){
        return redirect("/web/noteslist");
    }else{
        return redirect("/web/notesadd");
    }
}

// 验证插入笔记的内容
private function validator(array $data)
{
    return Validator::make($data, [
        'title' => 'required|max:255',
        'content' => 'required',
        // 'cid' => 'required',
    ]);
}

// 删除笔记
public function getDeletenote($id)
{
    $model = TbRecords::find($id);
    $model->state = TbRecords::$state['delete'];
    $cid = $model->cid;
    $model->save();
    $catModel = new TbCategory();
    $catModel->countReduce($cid,1);
    return redirect("/web/noteslist");
}

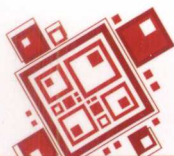
// 编辑笔记
public function getNoteedit($id)
{
    $model = new TbCategory();
    $catData = $model->findUserAll($this->getAttribute('uid'));
    $this->addAttributes('cglist',$catData);
    $model = TbRecords::find($id);
    $data = $model->toArray();
    $data['attributes'] = $this->attributes;
    return view('/backyard/notes/edit',$data);
}

public function postNotechange($id)
{
    $model = TbRecords::find($id);
    $model->fill($_POST);
    $model->save();
    return redirect("/web/noteslist");
}

```

```
public function getNoteshow($id)
{
    $model = TbRecords::find($id);
    $data = $model->getAttributes();
    $data['attributes'] = $this->attributes;
    return view('/backyard/notes/show',$data);
}
```

本章通过一个简单的实例将 Laravel 框架中各功能模块融合在一起完成一项简单的功能。在实现过程中可以看到，通过 Laravel 框架开发一个应用主要工作集中在前端界面、业务逻辑和数据库构建，这些工作量其实也不算少。Laravel 框架的优势是提供了一整套服务器程序组件，由于这些组件都是基础的、通用的，所以通过它们用户可以非常灵活、自由地开发一个项目，受到的约束很小；缺点是如果要实现一个项目还有大量的工作要做，所以 Laravel 更适合于企业级开发，有大型的团队做支持。如果需要快速开发一个应用或者自己创建一个应用，这个工作量可能也是无法承受的。所以，在一个项目中，是否采用 Laravel 框架还需视具体情况而定。



Laravel框架关键技术解析

对于用户的请求，Laravel就像流水线作业一样，通过一道道工序处理用户的请求，然后返回处理的结果。在这个过程中，用户可以很容易地增加、修改、删除其中的工序，实现定制化。

能够做到这些，笔者认为，主要是因为开发者在设计期间采用了组件化开发、依赖注入、接口编程等技术，组件化开发使得整个框架像搭积木一样构建起来，因此就可以非常容易地添加、删减功能，体现了编程技术中的易复用、可扩展等特性；依赖注入、接口编程使得模块间的耦合度非常低，如果想将某个模块替换自己新设计的模块，只需要满足接口规范就不会对其他模块产生影响，这体现了编程技术中的易维护特性。总之，通过学习Laravel框架，不仅可以掌握Web开发的方方面面，最重要的是能够学到构建一个优秀框架的思想和方法。

本书重点介绍了Laravel框架构建的关键技术，即组件化开发和使用的相關设计模式，所以本书适合想了解框架构建技术的读者。同时，本书是从源码层次分析该框架实现的几个方面，通过这些源码读者能了解实现的细节，从而很容易实现对该框架的定制和修改，并非只是简单的应用，通过掌握该框架的几个重要方面，读者能够在整体上把握Laravel框架实现的过程，所以本书适合想深入了解Laravel框架的读者。



博文视点Broadview



@博文视点Broadview

上架建议：网站开发>PHP

ISBN 978-7-121-29209-5



9 787121 292095 >

定价：79.00元



策划编辑：孙学瑛

责任编辑：徐津平

封面设计：李玲